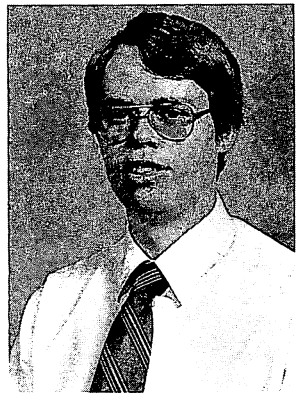
SAMS PRESENTS

22208

BASIC Tricks For The Apple

Allen Wyatt

BASIC Tricks for the Apple®



Allen Wyatt has been actively involved with the microcomputer industry for six years and is currently software development supervisor for Sams Software in Indianapolis, Indiana. Mr. Wyatt has had extensive experience in computer consulting and software development.

He has written several commercial software packages utilizing many of the same techniques detailed in BASIC TRICKS FOR THE APPLE. The broad range of computer programs runs the gamut from small system data bases to games and utilities.

Besides being a computer author, Allen is a devoted family man and active church member. He uses his personal Apple and IBM computers to assist him in all of these areas. At home, his family spends many hours using the computer every day.

BASIC Tricks for the Apple®

by

Allen Wyatt

Howard W. Sams & Co., Inc.
4300 WEST 62ND ST. INDIANAPOLIS, INDIANA 46268 USA

FIRST EDITION
FIRST PRINTING—1983

International Standard Book Number: 0-672-22208-6
Library of Congress Catalog Card Number: 83-50830

Edited by: *Patricia Perry*
Illustrated by: *Kevin Caddell*

Printed in the United States of America.

Preface

This book has been written with several purposes in mind. They are as follows:

1. To encourage programmers and computer-users to think logically.
2. To provide a source of efficient subroutines for the aspiring programmer.
3. To provide a fix for computer junkies.
4. To make money for the author.

Hopefully all of these will be realized. Before actually beginning, I would like to thank two people. First, Steve Wozniak for making his personal computer the Apple that tempted a generation. Second, my patient wife, Debbie, who has endured being a computer widow all these years. Perhaps in the near future they will discover a cure for computer addiction, but by then I hope I'm gone, because I certainly enjoy what I do.

ALLEN WYATT

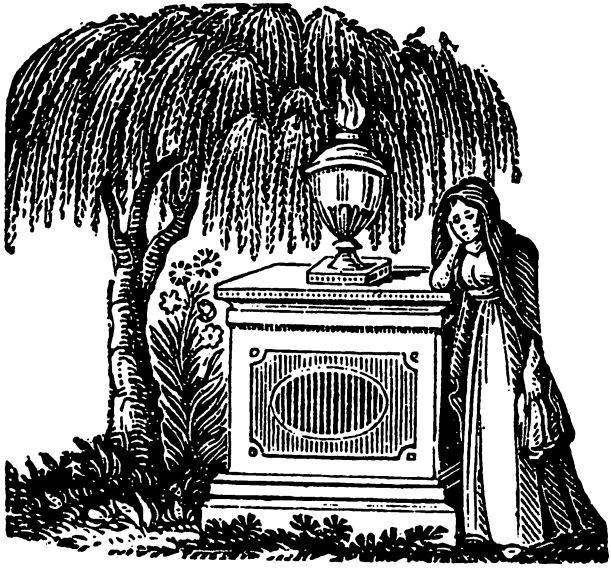


Fig. P-1. Computer widow, circa 1912 (pre-Apple).

Contents

CONTENTS

CHAPTER 6

CHAPTER 7

CHAPTER 8

CHAPTER 9

CHAPTER 10

CHAPTER 11

CHAPTER 12

Note

Throughout this book, reversed letters (e.g., RETURN) designate keystrokes. That is, when this designation is used, it means the reversed letters are a single keystroke on the keyboard (or keys that are pressed simultaneously to produce a desired result) instead of being input as individual characters. This same designation is also used for individual letters which are pressed to initiate a desired action in the program.

Introduction

One day I was browsing through the local offerings at the book store in my home-town mall, when I noticed something interesting. There were no books designed to help a computer user write BASIC routines that were useful, quick, and efficient.

Most all of the books said, in effect, "Here are the tools, BASIC, and your computer. They are great." But no one really showed how to make them do great things (with the possible exception of creating a few outstanding games of tic-tac-toe).

Hopefully, this book will go a long way to fill that gap. That is not to say that this book will be a cure-all, or even the cat's meow. Probably far from it. We may not see the cat even start to stir until the coming generation unleashes the power locked within their personal computers.

However, I feel that this book can be useful and, indeed, inspiring, if used in a proper way. What is the proper way? That is pretty much up to you. The book is written so that the reader can start just about anywhere, and have no problem understanding it. You can read it as a novel (if you enjoy that kind of reading) or you can use it as a reference manual.

This book will assume that you know BASIC. I know what they say about assuming, but that is where this book is aimed—at those who know BASIC, but just can't quite figure out how to make it perform all those neat tricks that would be so helpful on their Apple.

Hence the name *BASIC Tricks for the Apple®*. Since this book is written for the Apple in particular, it will be helpful if you have access to an Apple®* personal computer. Any old Apple will do, with the possible exception of an Apple I. The version of BASIC used will

---

*Apple is a registered trademark of Apple Company, Inc.

be Applesoft, and that is assumed to be in ROM. If you do not have access to an Apple, I am sure that your neighborhood computer dealer would just love for you to wander in and buy one or two. (Tell them I sent you. Then they will love me, too.)

I suppose that a lot of people, who could be referred to as "micro professionals," will poo-poo this book because they feel that BASIC is too slow, or that the graven image of machine code is more elegant or all-powerful. To them I can only say—Ha! 'Tis a sad day when we have nothing more to learn from those around us. BASIC is just fine for many everyday applications, and if we do what we do well, it doesn't matter what we use. All we need to do is use it well. It reminds me of the old argument over which language is easier—English or Spanish (or French, German, Chinese, or Tongan). There is no real answer. It all depends on what you grew up with.

Before we get to the main events, we should tell you what is on the program. The first chapter gives you a little background on the BASIC language, particularly Applesoft. Since many readers of this book may not know the history of Applesoft, they may find it interesting, although not essential to using BASIC effectively. (If my college teachers heard me say that, they would give me a retroactive "F" in my INTRO TO DATA PROCESSING class. Oh well, I guess the secret is finally out—you don't need to know the history of tools to know how to build a house.)

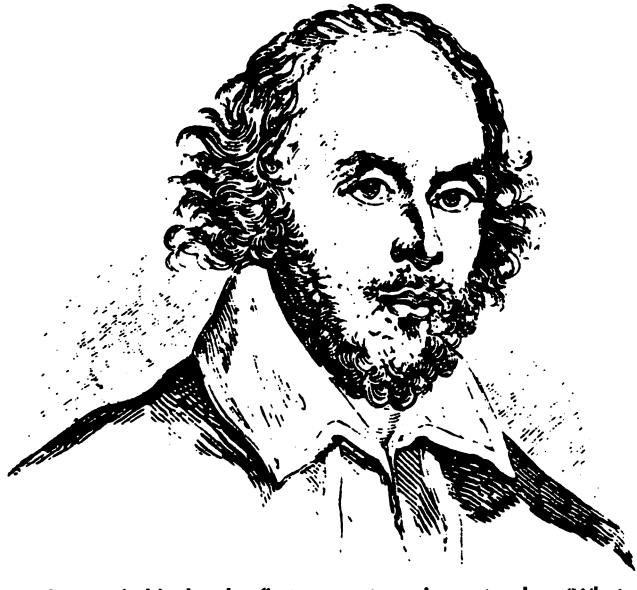


Fig. I-1. Abner Fritzbingle, the first computer science teacher (Whatsamatter U, March, 1918).

The other chapters of the book are written as self-contained lessons on various applications routines. They can be read in sequence or out of sequence.

Each subroutine and program listing in this book also includes variable tables. These are to aid you in converting these routines for use in your programs. The tables are included immediately following each listing.

Finally, it should be pointed out that there is a difference between working BASIC, effective BASIC, and efficient BASIC. Each is a new plateau that encompasses the one before, but does the same tasks faster and better. Hopefully this book will help you climb from one plateau to another.

CHAPTER 1

A Short Basic History

This "history" is not intended to be an in-depth study of the background of the BASIC, but rather a quick overview of the roots of the language. It includes highlights with a definite Apple flavor.

(Before actually beginning, I should mention my gratitude to Randy Wigginton. Randy provided helpful information and insight into the development of Applesoft. I greatly appreciate that.)

BASIC was developed at Dartmouth College, and made generally available by the year 1965. It was developed by Dr. John Kemeny and Dr. Thomas Kurtz primarily as a teaching language in a time-sharing environment.

BASIC is an acronym for *B*eginner's *A*ll-purpose *S*ymbolic *I*nstruction Code. It is a compilation of elements found in both FORTRAN and ALGOL. Since its introduction, the simple structure and plain-English commands of BASIC have made it the most used computer language in the world. It is used on all types of hardware, and in many different versions.

One of the big advantages of BASIC is that it is very "user-interactive." The language is interpreted at run time, meaning that each individual BASIC instruction is interpreted (parsed) when the program is actually run. Before BASIC, most languages needed to have their instructions compiled, or translated to machine language, in order to be understood by the computer. BASIC, by contrast, allows programmers to develop programs faster than had been possible in earlier languages. Today, several computer manufacturers have introduced both interpreter and compiler versions of BASIC for their machines. In this way a program can be developed quickly in BASIC and then compiled so that it will execute faster.

The simplicity and the popularity of BASIC has, however, led to such diversification that there are now many different dialects and

versions of the original language. Since there is no real "standard" for BASIC, each hardware vendor changes the language just a little so that it takes advantage of certain features of his computer.

In many ways, the de facto standard of BASIC is the Microsoft version of the language. In one form or another, the different versions of the language developed by this vendor have touched almost every personal computer around.

In the mid 1970s, Microsoft's 8K BASIC, originally written for an 8080 microprocessor, was converted for the 6800 and 6502 processors. Apparently versions of these translations were used by Commodore, MOSTech (KIM computer), and Apple at about the same time. The original version of Applesoft (called FP BASIC) was one of these translations.

When the Apple II was released to the general public in March of 1977, Integer BASIC was the only language shipped with the new computer. This was because it was the only completely "finished" language that was available to the manufacturer. It was compact, powerful, and fast, but it lacked the floating-point abilities that most computer users needed. Apple wanted to include a floating-point BASIC with that original Apple II, but it was not available by the computer's scheduled release date.

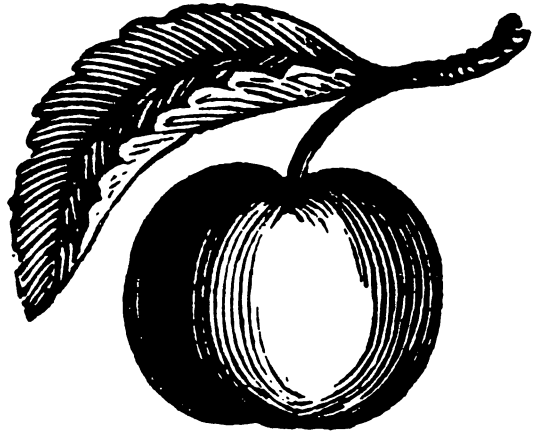


Fig. 1-1. Apple (the first).

The original "plan" called for Steve Wozniak and Randy Wigginton to develop a floating-point BASIC for the Apple II. Due to tight time schedules, however, this plan was never carried out. As an alternative, Apple approached Microsoft about using that company's 8K BASIC as a basis for Apple's machine. Microsoft sent the object code for the BASIC to Apple in July of 1977, and Randy began the conversion process.

Fig. 1-2. The Apple II. *(Courtesy Apple Computer, Inc.)*

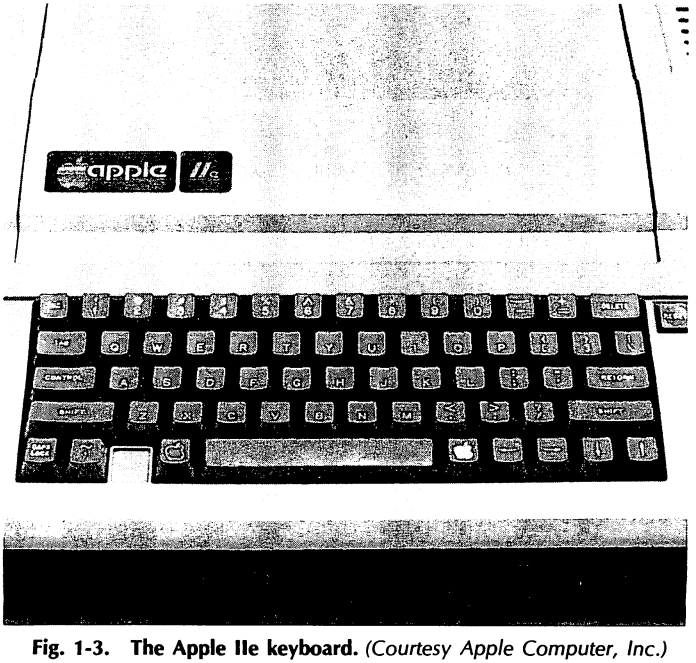


Fig. 1-3. The Apple IIe keyboard. *(Courtesy Apple Computer, Inc.)*

Fig. 1-4. The Apple III. *(Courtesy Apple Computer, Inc.)*

Several quick patches were made to this BASIC to allow for primitive graphics commands, as well as an initial "help" screen that explained language syntax when the program was first run. These changes expanded BASIC to a size of about 10K. The help screens were conceived and written by Mike Scott.

FP BASIC (Applesoft I) was released around September, 1977. Since there were no disk systems widely available as of that time, it was released on cassette as an Integer program to be loaded into lower RAM. This left the user with much less available memory since most Apples in those days did not have more than 16K of RAM.

Applesoft I, apparently released too soon, had numerous problems which caused it to bomb. Among the problems was the fact that the development was done from the object code, not the source code, of Microsoft's BASIC. Besides this, Apple's rush to get a floating-point

BASIC to market may have prompted them to release it before it was really ready.

About the same time that Applesoft I was released, however, Apple secured the source code to Microsoft's BASIC, and work on Applesoft II began.

The source code was transferred from the paper tape to a local time-sharing firm's computer for development under a cross-assembler. It took about a month to complete the transfer and make sure that it was bug-free. All of the new graphics commands were added, and Applesoft II took shape. It was polished and refined by December, 1977. Then disaster struck in the form of a system crash on the time-sharing system. Over two months worth of work was lost, and development had to start again with the October, 1977 version of Microsoft's BASIC.

A little gun-shy by now, Apple commenced development of Applesoft II anew on an in-house system, using a cross-assembler on a NorthStar computer. The improved language was finally finished and released on cassette in April, 1978.

Since that time, there have been no changes made to the code of Applesoft II. The version number was soon dropped, and the language is now simply referred to as Applesoft. There have been changes, however, in the media upon which the language is available. After the release of Applesoft on cassette, a ROM version was released around June 1978. Applesoft is currently available on ROM and disk (for use with the language system).

Even after the final introduction of Applesoft, the standard language for the Apple II was still Integer BASIC. All of this changed (apparently due to public pressure) with the introduction of the Apple II+ computer in 1979. The only real difference between the II+ and the II was the fact that Applesoft was available "on-board" the computer, instead of the original Integer BASIC. The Apple IIe (the "e" stands for "enhanced") was released in late 1982, and also has Applesoft as the on-board version of BASIC.

CHAPTER 2

Selective Input

If there is one common thread linking all programs, it is the fact that they need human input to make them valuable. For instance, what good would a payroll program be unless the paymaster let the computer know how many hours, and at what rate of pay, John Doe worked this last week?

But there are trade-offs everywhere. Just as human input makes a program valuable, it also opens up a veritable Pandora's box of errors and misunderstandings. Not everyone in the world thinks like the programmer, so they may have no idea what a computer really wants when it asks a question. For instance, consider the following short program:

```
10 INPUT "HOW MANY HOURS: ";H
20 INPUT "WHAT RATE OF PAY: ";R
30 P=H * R:PRINT "CORRECT PAY IS: ";P
```

Looks like a whiz-bang program, right? Wrong! At this very moment, there is probably some paymaster in Podunksville who would start to run this program and enter "STANDARD" as an answer to the first question. We wouldn't have to worry about the second question, however, because the program would stop functioning after the first answer.

What is the best way to handle this type of problem? The first and probably most important way is to avoid using numeric variables in inputs. Those can cause nasty problems. For instance, on the Apple, pressing RETURN when you are prompted for a numeric variable will cause an error since RETURN is not a number.

If we rewrite the previous program the new way, using string variables on the inputs, and then converting the strings to their number values, we will see this:

```
10 INPUT "HOW MANY HOURS: "; H$: H = VAL(H$)
20 INPUT "WHAT RATE OF PAY: "; R$: R= VAL(R$)
30 P = H * R:PRINT "CORRECT PAY IS: "; P
```

So far, so good. At least we won't get an error if our brilliant paymaster enters "STANDARD" for number of hours. However, we also won't get any pay. This is because the numeric value of "STANDARD" is zero, and, as we all know, zero multiplied by anything else is still zero.

Now comes the interesting part. How do we check to make sure that the user has input a valid entry? First, we have to follow a very simple set of guidelines:

1. Decide what input is wanted.
2. Develop a routine to get only that type of input.
3. Check to make sure the input is within predetermined acceptable bounds.
4. Test the heck out of it.
5. Let someone else test the heck out of it.
6. Never assume that the routine is foolproof, because some fool is bound to prove you wrong.

Simple, right? The first step is to determine what type of input we want: numeric or alphanumeric. We will discuss both types shortly, but first let's look at a very useful command.

THE GET STATEMENT

Applesoft BASIC has a neat little command which allows your computer to do wonderful things. This is the GET statement. There are, however, a few things to remember when using it. Always GET strings, never get numerics. For the reasons described before, numeric inputs (or GETs) can cause problems.

Let's assume the following line is being executed:

```
10 GET A
```

If the user types a number, everything is fine. However, if the user presses RETURN , then a SYNTAX ERROR is generated. Big problems, right? Well, there's more. Typing any alphanumeric key in response to the GET statement above will return a zero value in A, unless the user presses a colon or comma. When that happens, the message ?EXTRA IGNORED is generated, and all is not well in Podunksville.

Thus the simple GET rule: Always get strings. When getting a string, you can press almost anything without adverse results. I say almost anything because someone is bound to push RESET to see

what will happen. Well, I can tell you. The speaker will beep and you will be returned to command mode. This is because RESET is not an ASCII character; it is a hardware interrupt.

NUMERIC INPUT ROUTINES

By using the GET statement, the following can become the basis of our numeric input routine:

```
110 GET A$: A$ = LEFT$(A$,1)
120 IF (A$ < "0" OR A$ > "9") AND A$ <> "." THEN 110
130 PRINT A$;
```

This set of instructions does quite a bit in just three lines. First of all, line 110 gets a character from the user and makes sure it got only one character. Under certain obscure conditions (usually only when the moon is full on the 5th Thursday in February) Applesoft will return two characters with a GET statement. Supposedly this occurrence has something to do with the dreaded malady of "keybounce," meaning that the computer interprets one keypress as two keypresses of the same key in quick succession. This possible problem is circumvented by the instructions in line 110. As the saying goes, it is better to be safe than sorry. The second part of line 110 makes sure we are safe.

Line 120 checks the character to make sure the user has entered a valid number or decimal point. Thus, if a letter or other symbol is pressed, the keypress is ignored, and the computer goes back to waiting for another key to be pressed.

Finally, if the keypress was acceptable, line 130 prints what was typed. Notice the semicolon at the end of the statement. This ensures that all of the characters we are printing stay on the same line.

Now, we need to add a few more lines to do some other necessary tasks. The answer being typed by the user must be accumulated into a handy place. We must also be able to check to see if the user typed a RETURN, or if the first character that the user typed was a negative sign. With these features, our routine now looks like that of Listing 2-1. The variables used in this routine are summarized in Tables 2-1 and 2-2.

Notice our fundamental routine in there? All we did was add an initialization line to clear the variables that we will be using in the routine (line 100). Then, in line 115, we checked to see if the last keypress was the RETURN key. If it was, we jumped to line 150 where we assign to variable 'B' the actual value that was input, then return from the routine.

Line 116 and 117 check to see if the user wants to go back a space. After all, everyone makes mistakes, so CHR$(8) is a backspace. If it

```
100  A$ = "":B$ = "":C$ = "":B = 0:D = 0

105  IF C$ = "." THEN D = 0:C$ = ""

110  GET A$:A$ = LEFT$ (A$,1)

115  IF A$ = CHR$ (13) THEN 150

116  IF A$ = CHR$ (8) AND LEN (B$) > 0 THEN C$ = RIGHT$
      (B$,1):B$ = LEFT$ (B$,LEN (B$) - 1):PRINT A$;" ";
     A$;

117  IF A$ = CHR$ (8) THEN 105

118  IF B$ = "" AND A$ = "-" THEN 130

120  IF (A$ < "0" OR A$ > "9") AND A$ <  > "." THEN 110

125  IF A$ = "." AND D = 1 THEN 110

130  PRINT A$;

135  IF A$ = "." THEN D = 1

140  B$ = B$ + A$: GOTO 110

150  B = VAL (B$):RETURN
```

Listing 2-1

Table 2-1. Entry and Exit Variables for Numeric Input Routine

On Entry	On Exit
NONE	B$—String entered by user B—Numeric value of string

Table 2-2. Variable Table for Numeric Input Routine

Variable	Type	Purpose	Lines used in
A	String	Keypress character	100, 110, 115, 116, 117, 118, 120, 125, 130, 135, 140
B	String	User input	100, 116, 118, 140, 150
B	Numeric	String value	100, 150
C	String	Backspace character	100, 105, 116
D	Numeric	Decimal Flag	100, 105, 125, 135

was pressed, line 116 sets C$ equal to the character we are to subtract, and then the rightmost character of B$ is subtracted. A backspace (A$) is then printed, followed by a space to blank out the last character on the screen, and then another backspace (A$) to get ready for the next keypress. Fig. 2-1 shows this process. Line 117 then jumps back to line 105. That line will be explained in a moment.

Notice that if B$ had nothing in it when the left arrow was pressed, then all of the above manipulation is skipped, and line 117 just takes us back to line 105 without the string manipulation.

Line 118 checks to see if the character pressed was a minus sign. If it was, and only if it is going to be the first character of the entry, it is accepted. Notice that B$ will be null (equal to nothing) only if we are analyzing the first character pressed.

A routine was also included to check to see that only one decimal point was entered in the number. Line 125 and 135 respectively check and set the flag if a decimal point has been entered. A decimal point will be allowed only if D is still zero. Once that one decimal point has been entered, D is equal to one, and we continue, accepting numbers only.

We are talking about getting only one decimal point. But what would happen if the user typed a decimal point, followed by a number, then backspaced two places? Without a proper way to check, he would not be able to enter the decimal point again, be-

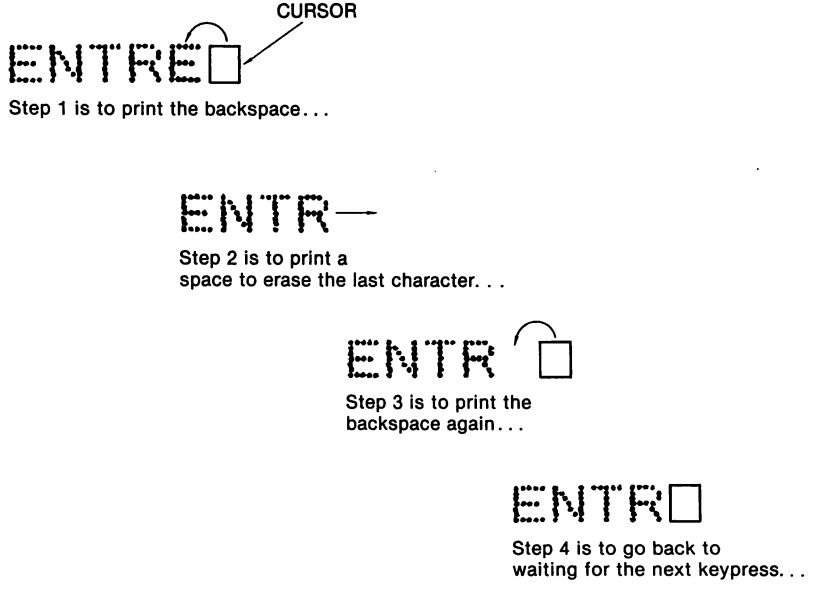


Fig. 2-1. The input routine backspace process.

cause the D flag would still be set to one. That is why when a backspace is pressed, we always loop back to line 105. This checks to see if the character being backspaced over, and thus deleted from the entry, is a decimal point. If it is, D is set to zero so that another decimal point can be accepted.

Finally, line 140 adds the keypress to B$. Then we go back to get another keypress at line 110.

Notice that the whole idea behind this routine is to accept only a numeric input. Only a negative sign and any legal number or decimal point (only one) will be accepted.

Will this work under all circumstances? Probably not, because users can be very resourceful in finding illegal entries to make programs bomb. However, it is a great deal more reliable and sophisticated than the normal INPUT statement.

There are a few other statements you could add to this routine to tailor it to your specific needs. For instance, you could add a statement at the end of the routine to check to make sure that the input is within limits. Suppose you do not want a number smaller than 0, or larger than 999.99. If this is the case, you can naturally take out the checks for a negative sign, and then check to make sure that the length of B$ never goes beyond six characters.

How will the numeric input routine fit into the payroll program that we started to develop earlier? Listing 2-2 gives this program and Table 2-3 lists the variables in this program.

Notice that we have renumbered the early part of our program and added it to the end of our subroutine. This was done not to confuse you, but to conform to a standard required of those who want to write good, efficient BASIC code. All common subroutines should be located near the beginning of the program, with all controlling routines residing after them. Believe it or not, this actually helps the program execute faster.

ALPHANUMERIC INPUT ROUTINES

Now let's talk a bit about alphanumeric input. Again, we could use the standard INPUT statement, but that wouldn't allow for several things. First, the user can't put in commas or colons. If he does, then all he succeeds in doing is getting an error message stating ?EXTRA IGNORED. This can be disconcerting, at the least.

Also, using the INPUT statement makes it difficult to weed out control characters. For instance, if our experienced paymaster was entering a person's name into his program, and accidentally pressed a CTRL-F in the middle of a name, what do you think would happen when that person's name was printed on a check or on a report? Surprise! A CTRL-F is a form-feed on most printers. Your baffled (not

```
90    GOTO 1000
100   A$ = "":B$ = "":C$ = "":B = 0:D = 0
105   IF C$ = "." THEN D = 0:C$ = ""
110   GET A$:A$ = LEFT$ (A$,1)
115   IF A$ = CHR$ (13) THEN 150
116   IF A$ = CHR$ (8) AND LEN (B$) > 0 THEN C$ = RIGHT$ (B$,1):B$
       = LEFT$ (B$,LEN (B$) - 1):PRINT A$;" ";A$;
117   IF A$ = CHR$ (8) THEN 105
118   IF B$ = "" AND A$ = "-" THEN 130
120   IF (A$ < "0" OR A$ > "9") AND A$ <  > "." THEN 110
125   IF A$ = "." AND D = 1 THEN 110
130   PRINT A$;
135   IF A$ = "." THEN D = 1
140   B$ = B$ + A$: GOTO 110
150   B = VAL (B$):RETURN
1000  PRINT "HOW MANY HOURS: ";:GOSUB 100:H = B
1010  PRINT "WHAT RATE OF PAY: ";:GOSUB 100:R = B
1020  P = H * R:PRINT "CORRECT PAY IS: ";P
9999  END
```

Listing 2-2

Table 2-3: Variable Table for Payroll Program

Variable	Type	Purpose	Used in Lines
A	String	Keypress character	100, 110, 115, 116, 117, 118, 120, 125, 130, 135, 140
B	String	User input	100, 116, 118, 140 150
B	Numeric	String value	100, 150
C	String	Backspace character	100, 105, 116
D	Numeric	Decimal Flag	100, 105, 125, 135
H	Numeric	Hours	1000, 1020
P	Numeric	Pay	1020
R	Numeric	Rate	1010, 1020

to mention irate) paymaster would think his computer and/or printer was malfunctioning. You could try to explain that it was a simple case of operator error (heaven forbid any computer operator should ever make an error!), but it would be far simpler to disallow any such entries in the first place.

Another minus for the INPUT statement: you can't prevent the user from typing in an entry which is too long. Wouldn't it be great if a bell rang every time the user attempted to type characters beyond the legal length for the entry? What if all characters beyond that point could be ignored?

Well, we can do all of this with minor modifications to our numeric input routine. We are using the same basic instructions, but with slightly different input checks. The changed routine will look like the one in Listing 2-3. Tables 2-4 and 2-5 list the variables used.

Notice that this routine is a lot simpler than the numeric routine that we used earlier. That is because we are able to accept a much wider range of characters than before.

```
200  A$ = "":B$ = ""

210  GET A$:A$ = LEFT$ (A$,1):A = ASC (A$)

220  IF A = 13 THEN PRINT :RETURN

230  IF A = 8 AND LEN (B$) > 0 THEN B$ = LEFT$ (B$,LEN
     (B$) - 1):PRINT A$;" ";A$;

240  IF A = 8 THEN 210

250  IF A < 32 OR A > 90 THEN 210

260  PRINT A$;

270  B$ = B$ + A$: GOTO 210
```

Listing 2-3

We also have introduced a new concept in this routine. Since Applesoft deals faster with numbers than it does with strings, we have converted our keypressed character to a number with the ASC function in order to do all of our checks. This was done in line 210. Then all we have to do is check for a RETURN or left arrow by their ASCII equivalents and make sure the character pressed was within the range of valid, printable characters. In line 250 there is a check to make

Table 2-4. Entry and Exit Variables for Alphanumeric Input Routine

On Entry	On Exit
NONE	B$—User input string

Table 2-5. Variable Table for Alphanumeric Input Routine

Variable	Type	Purpose	Used in Lines
A	String	Keypress character	200, 210, 230, 260, 270
A	Numeric	ASCII value of keypress	210, 220, 230, 240, 250
B	String	Input string	200, 230, 270

sure A is not less than 32 (ASCII "space") or greater than 90 (ASCII "Z"). If it is, then the character is ignored, and the program branches to line 210 to get the next keypress.

If you need any help with ASCII values, check your Applesoft manual, and you can see what they signify on an Apple.

So far, we have weeded out all of the control characters, and we are allowing the user to enter commas, colons, quotes, and other printable characters. All we are doing is getting the keypress, checking it, and adding it to our input string. It would be nice, however, to put a length check in.

This is done by simply adding one line of code, as follows:

```
245 IF LEN(B$)=UL THEN PRINT CHR$(7);:GOTO 210
```

This line ensures that the length of the entry doesn't go beyond a predetermined boundary. The length should be specified by setting UL to any desired "upper limit" before entering the routine. Once this limit has been reached, if the user tries to enter any character except a backspace or a carriage return, then the bell will ring, and the keypress will be ignored.

Let's put this new input routine to work in our payroll program. (See Listing 2-4 and Table 2-6.)

Now, if our Podunksville Paymaster enters a name as "DOE, JOHN", there will be no problem. The routines allow for this without generating error messages to confuse the user.

INPUT CONCLUSIONS

A nice thing about these routines is that they can be used with disk files. If you write information to a disk file, you can later use this input routine to retrieve it. All you need to do is call the routine instead of using the familiar INPUT statement.

In the beginning of the chapter, we mentioned the constant trade-offs that are part of working with computers. This is a good place to give an example of one of those trade-offs.

```
90    GOTO 1000

100  A$ = "":B$ = "":C$ = "":B = 0:D = 0

105  IF C$ = "." THEN D = 0:C$ = ""

110  GET A$:A$ = LEFT$ (A$,1)

115  IF A$ = CHR$ (13) THEN 150

116  IF A$ = CHR$ (8) AND LEN (B$) > 0 THEN C$ = RIGHT$ (B$,1):B$
      = LEFT$ (B$,LEN (B$) - 1):PRINT A$;" ";A$;

117  IF A$ = CHR$ (8) THEN 105

118  IF B$ = "" AND A$ = "-" THEN 130

120  IF (A$ < "0" OR A$ > "9") AND A$ <  > "." THEN 110

125  IF A$ = "." AND D = 1 THEN 110

130  PRINT A$;

135  IF A$ = "." THEN D = 1

140  B$ = B$ + A$: GOTO 110

150  B = VAL (B$):RETURN

200  A$ = "":B$ = ""

210  GET A$:A$ = LEFT$ (A$,1):A = ASC (A$)

220  IF A = 13 THEN RETURN

230  IF A = 8 AND LEN (B$) > 0 THEN B$ = LEFT$ (B$,LEN (B$) - 1):
     PRINT A$;" ";A$;

240  IF A = 8 THEN 210

250  IF A < 32 OR A > 90 THEN 210

260  PRINT A$;

270  B$ = B$ + A$: GOTO 210

1000 PRINT "EMPLOYEE NAME: ";:UL = 20:GOSUB 200:N$ = B$

1010 PRINT "HOW MANY HOURS: ";:GOSUB 100:H = B

1020 PRINT "WHAT RATE OF PAY: ";:GOSUB 100:R = B

1030 P = H * R:PRINT "CORRECT PAY FOR ";N$;" IS ";P

9999 END
```

Listing 2-4

Since we have gained so much latitude on user input, we have to give up something somewhere, right? Right! By using an input routine such as this, we lose a bit of speed. The user won't usually notice it when he is typing in an entry, but the slowdown is there, nonetheless.

Table 2-6. Variable Table for Payroll Program

Variable	Type	Purpose	Used in Lines
A	String	Keypress character	100, 110, 115, 116, 117, 118, 120, 125, 130, 135, 140, 200, 210, 230, 260, 270
A	Numeric	ASCII value of keypress	210, 220, 230, 240, 250
B	String	User input	100, 116, 118, 140, 150, 200, 230, 270, 1000
B	Numeric	String value	100, 150
C	String	Backspace character	100, 105, 116
D	Numeric	Decimal Flag	100, 105, 125, 135
H	Numeric	Hours	1010, 1030
N	String	Employee name	1000, 1030
P	Numeric	Pay	1030
R	Numeric	Rate	1020, 1030

Because so much string manipulation is being done with each keypress, your Apple will have to do "garbage collection" more often to clear out unwanted strings. This shouldn't be a big problem, unless your program is very long or uses quite a bit of memory.

So much for the trade-off. In our next chapter we will look at how to round—not circles, but numbers. You may finally discover why your school teachers stressed math.

CHAPTER 3

Rounding

Rounding is the basic art of making numbers come out to the nearest "something." Rounding is usually used in relation to money figures, and that "something" to which you round can be the nearest dollar, penny, or fraction of a penny.

Other uses for rounding, though, are abundant. You may need to round a number to the nearest hundred for a calculation. Or, when calculating disk space from within a configuration program, you may need to "round down" to make sure that there is enough room on the disk for the records wanted.

The technique of rounding is a rather simple one. The basic rounding equation looks like this:

```
N = INT ( N / D + K ) * D
```

Each item in this formula deserves a little individual attention. As you notice, there are only three variables used; N, D, and K.

The variable N is the number that we want rounded. It should be set to a value before entering this formula, and after exiting, it will be equal to the rounded value that we want.

The variable D is called the divisor / multiplier because it serves as both at different times within the equation. This variable determines to how many decimal places, or to what "precision" N will be rounded. We will look at the effect of various numbers in this location in a moment.

The variable K is the "kicker." That sounds like a descriptive term, doesn't it? Well, that is exactly what it does. It "kicks" our number either up, down, or to the nearest figure. Its effect will be seen in our later examples.

If we were using this formula in a program, we would first need to decide to how many places we wanted to round our numbers. If we

were working only with dollars and cents, we would round our figures to two decimal places. This, again, determines what we would use for the variable D in our formula.

Some common values for D, based on rounding to powers of 10, are shown in Table 3-1.

For dollars and cents, we would use the value .01 for D. For the value K, there are usually only 3 different values used when rounding to powers of 10. When K is equal to 1, we would be rounding up. If K was equal to .5, we would be rounding to the nearest figure. If K was equal to 0, then we would be rounding down. Notice that if K was equal to 0, then it could be left out of the equation completely.

ROUNDING TO THE NEAREST FIGURE

If we wanted to round a dollar and cents figure to the nearest penny, we could use the following line of code:

```
100 N = INT ( N / .01 + .5 ) * .01
```

Instead of using a line of BASIC code similar to that above, we could just as easily have made use of the DEF FN statement of BASIC. This would have allowed us to define our rounding formula as an integral function, as follows:

```
100 DEF FN A ( N ) = INT ( N / .01 + .5 ) * .01
```

The advantage that may be found here is that you can later call this by a simple BASIC function statement. For instance, it may look like this:

```
200 A = FN ( A )
```

Both routines, whether used as callable subroutines, or as independent functions, will round your number for you. If you follow through on the logic, the routine takes a number that you "feed" to it (within

Table 3-1. Rounding Divisors & Multipliers

Divisor/ Multiplier	Result Rounded to Nearest:
0.0001	4 decimal places
0.001	3 decimal places
0.01	2 decimal places
0.1	1 decimal place
1	whole number
10	ten
100	hundred
1000	thousand
10000	ten thousand

the variable "N"), and then rounds it. Perhaps it is best to show an example—one which may remind you of your high school Algebra class.

First we will need an unrounded dollar figure, such as 157.19832. When this value is substituted for the variable "N", our equation looks like this:

```
N = INT (157.19832 / .01 + .5) * .01
```

Next, by performing the calculation within the parentheses (remember that precedence within the computer determines that any divisions will be done before additions), our formula will look like this:

```
N = INT (15720.332) * .01
```

Next, the INT function will be executed. This function will truncate all of the numbers following the decimal point of the argument value. So, during the next step, the formula looks like this:

```
N = 15720 * .01
```

Finally, the last operation involves multiplying the number by .01. Notice that the answer will be:

```
N = 157.2
```

Upon completion, our number is rounded to the nearest penny. This is great for printing dollar figures without trailing fractions of pennies that only the pickiest accountants would be interested in anyway. In a later chapter, we will discuss ways of taking our rounded numbers and formatting the output so that they look nice and neat.

ROUNDING UP

Using our dollars and cents example, we can see how easy it is to round up. To do this the statement would be:

```
100 N = INT ( N / .01 + 1 ) * .01
```

This routine will always round a number to the next highest penny. (It's great for bankers figuring interest due.) The proof (with our previous example) works like this:

```
N = INT ( 157.19832 / .01 + 1 ) * .01
N = INT ( 15719.832 + 1 ) * .01
N = INT ( 15720.832 ) * .01
N = 15720 * .01
N = 157.2
```

It works fine with this particular example, but how does it work with a number that would normally have been rounded down? Let's examine it with the number 12.00001103 in the formula:

```
N = INT ( 12.00001103 / .01 + 1 ) * .01
N = INT ( 1200.001103 + 1 ) * .01
N = INT ( 1201.001103 ) * .01
N = 1201 * .01
N = 12.01
```

As before, our figure, although just slightly over $12.00 to begin with, was rounded to the next highest cent.

ROUNDING DOWN

Rounding down is the same procedure as rounding up, except there is no "kicker" added within the formula. The kicker in the above equation was the number 1 which was added to force a round up. With no kicker, the line of BASIC code to round down would look like this:

```
100 N = INT ( N / .01 ) * .01
```

As a proof, let's use our original number, since that number is usually rounded up:

```
N = INT ( 157.19832 / .01 ) * .01
N = INT ( 15719.832 ) * .01
N = 15719 * .01
N = 157.19
```

Notice that the number has been rounded down to the next lowest penny.

So far, we have been talking about rounding to a penny, whether it be nearest, up, or down. This is also referred to as rounding to two decimal places. You can round to more (or fewer) decimal places by simply increasing (or reducing) the number of zeros in the divisor / multiplier.

ODD ROUNDING

In our examples so far, we have rounded only to powers of 10. We could just as easily round to a different value to achieve different results. For instance, you may wish to round to the nearest 50, or to the nearest 25, or to the nearest 30. The number to which you round is entirely up to you. Just use different numbers in the formula.

If you want to experiment with different values for rounding, try the following short program in Listing 3-1. (The variables are listed in Table 3-2.) It will run through a few quick rounding formulas for you.

```
10  FOR J = 1 TO 100 STEP .26
20  RESTORE :FOR K = 1 TO 6:READ D
30  N1 = INT (J / D + 1) * D
40  N2 = INT (J / D + .5) * D
50  N3 = INT (J / D) * D
60  PRINT " NUMBER: ";J
70  PRINT " FACTOR: ";D
80  PRINT "     UP: ";N1
90  PRINT "NEAREST: ";N2
100 PRINT "   DOWN: ";N3
110 PRINT :NEXT K
120 PRINT :PRINT :NEXT J
130 END
200 DATA  .001, .01, .1, 3, 10, 11
```

Listing 3-1

Table 3-2. Variable Table for Rounding Example

Variable	Type	Purpose	Used in Lines
D	Numeric	Unrounded number	20, 30, 40, 50, 70
J	Numeric	Loop counter	10, 30, 40, 50, 60, 120
K	Numeric	Loop counter	20, 110
N1	Numeric	Rounded number	30, 80
N2	Numeric	Rounded number	40, 90
N3	Numeric	Rounded number	50, 100

When you run the program, it will go by on your screen rather quickly, so you may wish to have your fingers ready on the CTRL-S keys to stop output while you examine it. If this type of program really impresses you, you can add a statement in line 5 to make output go to your printer so that you will have a permanent copy of a program showing various numbers rounded in various ways.

The program can also be changed by using different step values in line 10, or by using different DATA elements to experiment with rounding capabilities.

As a final note, remember that even computers are not all perfect. The Apple does strange things with some numbers, as you may notice on your output. Because of the computer's internal representation of some numbers, they are never quite derived in the expected manner. Do not despair! We can all learn to work within the limits of a general-purpose computer.

CHAPTER 4

Dollars and Cents

How do we make sense of dollars and cents? What a perplexing question. In this chapter, we answer that question by showing you how to present your numeric output in a professional-looking manner.

First of all, in case you haven't noticed, numbers come in all shapes and sizes. There are big numbers (1.5623 E 23), small numbers (1.5623 E −23), and some in between (0). All of these numbers are useful, and indeed necessary, for a variety of applications, but they must be formatted to give a professional appearance on reports, as well as on the video screen.

The biggest problem is caused by all of those numbers to the right of the decimal point. When your decimal points are not in line, it looks like a chicken tracked all over your report. This can be very disconcerting, but there is a way to solve the problem.

The first rule of printing numbers is to not print numbers—print only strings. It is far easier to format and position strings than numbers. For instance, the following program prints only numbers. Try running it, then take a look at your output:

```
100 FOR J * 1 to 50 STEP 5.25
110 PRINT J
120 NEXT
```

This program prints output all over the place! Fig. 4-1 shows what you should have seen on your video screen. We got all of the numbers that we asked for, but they would have been much more readable if they were lined up in standard column order. There is no intrinsic way to accomplish this on the Apple.

```
]RUN

1
6.25
11.5
16.75
22
27.25
32.5
37.75
43
48.25

]
```

Fig. 4-1. Sample video screen.

FORMATTED OUTPUT

The first step in creating a formatting routine is to convert all numbers to strings, and then print the strings. The strings should all be of a uniform length, and each number should be right-justified within the string. This is not as difficult as it may sound. Consider our program with the subroutine in Listing 4-1 and its variables in Tables 4-1 and 4-2 which does all of this.

Notice that we also included a rounding function at line 10 (see Chapter 3) so that our numbers could be rounded to no more than two decimal places. When this program is run, your screen appears about the same as in Fig. 4-2. Notice that the figures are lined up on the right, and that each figure ends in the tenth column on the screen.

```
5     GOTO 100
10    N = INT (N / .01 + .5) * .01
20    A$ = RIGHT$ ("          " + STR$ (N),10)
30    RETURN
100   FOR J = 1 TO 50 STEP 5.25
110   N = J:GOSUB 10:PRINT A$
120   NEXT
```

Listing 4-1

Table 4-1. Entry and Exit Variable Table for Dollars and Cents Routine Example

On Entry:	On Exit:
N—Number to be rounded and formatted	A$—Formatted string N—Rounded number

Table 4-2. Variable Table for Dollars and Cents Routine Example

Variable	Type	Purpose	Used in Lines
A	String	Formatted Output	20, 110
J	Numeric	Loop Counter	100, 110
N	Numeric	Work Number	10, 20, 110

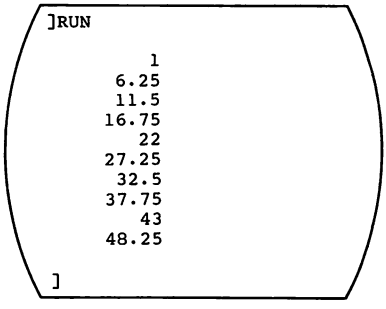


Fig. 4-2. Sample video screen.

This is a very good start, but the numbers are still not lined up by decimal points. This is because each number does not have the same number of digits to the right of the decimal point. We need a way to make sure that trailing zeros are included in each number so that two decimal places are showing.

Let's add another couple of lines to our routine to straighten this out. (See Listing 4-2 and Tables 4-3 and 4-4.)

```
5     GOTO 100
10    N = INT (N / .01 + .5) * .01
15    A$ = STR$ (N):IF N = 0 THEN A$ = ""
20    IF INT (N) = 0 THEN A$ = "0" + A$
25    IF N = INT (N) THEN A$ = A$ + ".00": GOTO 35
30    IF ASC (RIGHT$ (A$,3)) <  > 46 THEN A$ = A$ + "0"
35    A$ = RIGHT$ ("          " + A$,10)
40    RETURN
100   FOR J = 1 TO 50 STEP 5.25
110   N = J:GOSUB 10:PRINT A$
120   NEXT
```

Listing 4-2

Table 4-3. Entry and Exit Variable Table for Dollars and Cents Routine Example

On Entry:	On Exit:
N—Number to be rounded and formatted	A$—Formatted string N—Rounded number

Table 4-4. Variable Table for Dollars and Cents Routine Example

Variable	Type	Purpose	Used in Lines
A	String	Formatted Output	15, 20, 25, 30, 35, 110
J	Numeric	Loop Counter	100, 110
N	Numeric	Work Number	10, 15, 20, 25, 110

We have now added a few lines to make the routine more useful. Line 15 converts N to a string (A$), and then checks to see if N is a zero. If it is, then A$ is set to nothing so that all of the following manipulations will work correctly.

Next, in line 20, the program checks to see if N is between zero (inclusive) and one. If it is, then a leading zero is added to our string. In line 25, we check to see if N is a whole number. If it is, then our job is simple. All that needs to be done is to add the decimal point and two zeros. Then execution skips to line 35 where the string is filled out to 10 spaces.

If N is not a whole number, we should have to add only one zero. However, the number may already have two significant digits. It would be counter-productive to add a zero to a number that already is justified to two decimal places. Line 30 checks to see if the third character from the right is a decimal point. The ASCII value for a decimal point is 46 (page 138, Applesoft Manual). If it is a decimal point, then the test fails, and execution drops to line 35. However, if there is only one significant digit, then a zero is added to the end of the string to make sure that there are two places to the right of the decimal point.

As you probably already noticed, all routine execution goes through line 35 which adds enough spaces to the left of the string to make sure that it is 10 characters long. This has the effect of right-justifying the number within the string.

When this program is run, our output looks columnar. And isn't that what we wanted in the first place? This is shown in Fig. 4-3.

For most applications, you may be able to use this routine with no modifications. However, if you feel that you may be processing numbers greater than 9999999.99, then you may wish to make your formatted strings longer than 10 characters. This is easily done by changing line 35 to reflect a bigger string size than 10.

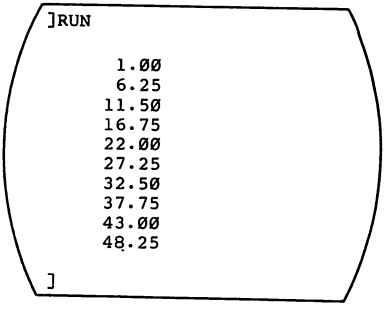


Fig. 4-3. Sample video screen.

Other special cases are processing negative numbers, or formatting numbers which include commas. These are addressed below.

FORMATTING NEGATIVE NUMBERS

Negative numbers can be formatted in one of two ways—with the negative sign either preceding or trailing the number. Both methods are acceptable for reports, and the decision of which method to use is entirely up to you and your intended program audience.

In order to format with a preceding negative sign, the above routine will work just fine, as long as the number, including the negative sign, does not go over 10 places. When we convert the number to a string, the sign is converted as well. By using this routine with no modification, the largest negative number we could use would be −999999.99. Again, if you need to allow for larger negative numbers, simply change line 35 to reflect a larger string.

If trailing negative signs are desired, then a change must be made to this routine. This is done by the use of a sign string, and an absolute value of the number to be formatted, as shown in Listing 4-3 and Tables 4-5 and 4-6.

The only changes to our routine were adding line 12, and changing line 35. Line 12 sets a negative sign equal either to a space, or to a minus sign, depending on whether N is less than 0 or not. If N is less than zero, then it is changed to a positive number for the remainder of the calculations. Notice this is done by use of the ABS function.

Finally, in line 35, the sign string is appended to our formatted string. Again, if the number is negative, it will have a trailing minus sign.

As a side note, notice that we reduced the size of the number that could be successfully formatted by one digit in line 35. This is to allow for the sign string to fill up the format to 10 characters. It is just as easy to use any other size string by changing the number 9 to some other number.

If we were to run this routine, the output would look similar to Fig. 4-4.

```
5     GOTO 100
10    N = INT (N / .01 + .5) * .01
12    NS$ = " ":IF N < 0 THEN NS$ = "-":N = ABS (N)
15    A$ = STR$ (N):IF N = 0 THEN A$ = ""
20    IF INT (N) = 0 THEN A$ = "0" + A$
25    IF N = INT (N) THEN A$ = A$ + ".00": GOTO 35
30    IF ASC (RIGHT$ (A$,3)) <  > 46 THEN A$ = A$ + "0"
35    A$ = RIGHT$ ("            " + A$,9) + NS$
40    RETURN
100   FOR J =  - 25 TO 25 STEP 5.25
110   N = J:GOSUB 10:PRINT A$
120   NEXT
```

Listing 4-3

Table 4-5. Entry and Exit Variable Table for Dollars and Cents Routine Example

On Entry:	On Exit:
N—Number to be rounded and formatted	A$—Formatted string N—Rounded number

Table 4-6. Variable Table for Dollars and Cents Routine Example

Variable	Type	Purpose	Used in Lines
A	String	Formatted Output	15, 20, 25, 30, 35, 110
J	Numeric	Loop Counter	100, 110
N	Numeric	Work Number	10, 12, 15, 20, 25, 110
NS	String	Sign String	12, 35

FORMATTING WITH COMMAS

Commas have a good side and a bad side in formatted numbers. On the good side, they look great on some reports. On the bad side, they can be a real bear to use, and depending on the size of the numbers being formatted, they can slow down the processing because of the amount of string manipulation they require.

To use commas, it would be best to drastically alter our subroutine

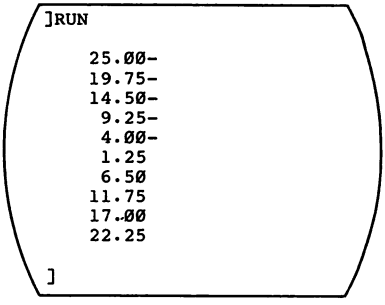


Fig. 4-4. Sample video screen.

to allow for separate processing of the decimal and whole parts of the number. This makes it quite a bit easier to insert commas. The subroutine, when written as in Listing 4-4, with the variables defined in Tables 4-7 and 4-8, does this rather nicely:

This is the super-duper do-it-all routine that inserts commas, justifies, and adds trailing negative signs if necessary. Line 35 handles adding commas by successively tearing apart the whole string into little three digit parts and adding them to the target string. Finally, in line 45, the final formatting is completed.

Because we are still only limiting our output to a total of nine absolute digits (the tenth is used for the sign), then the absolute value of

```
5     GOTO 100

10    N = INT (N / .01 + .5) * .01

15    NS$ = " ":IF N < 0 THEN NS$ = "-":N = ABS (N)

20    N1 = INT (N):N2 = N - N1

22    W$ = STR$ (N1):W1$ = "":IF N1 = 0 THEN W$ = "0"

25    D$ = STR$ (N2):IF N2 = 0 THEN D$ = ".00"

30    IF LEN (D$) < 3 THEN D$ = D$ + "0"

35    IF LEN (W$) > 3 THEN W1$ = "," + RIGHT$ (W$,3) + W
      1$:W$ = LEFT$ (W$,LEN (W$) - 3): GOTO 35

40    W1$ = W$ + W1$

45    A$ = RIGHT$ ("           " + W1$ + D$,9) + NS$

50    RETURN

100   FOR J =  - 10000 TO 10000 STEP 1500.25

110   N = J:GOSUB 10:PRINT A$

120   NEXT
```

Listing 4-4

Table 4-7. Entry and Exit Variables for Dollars and Cents with Commas Routine

On Entry:	On Exit:
N—Number to be formatted	A$—Formatted output N—Rounded number

Table 4-8. Variable Table for Dollars and Cents with Commas Routine

Variable	Type	Purpose	Used in Lines
A	String	Formatted output	45, 110
D	String	Temporary string	25, 30, 45
J	Numeric	Loop Counter	100, 110
N	Numeric	Work number	10, 15, 20, 110
N1	Numeric	Integer value	20, 22
N2	Numeric	Integer value	20, 25
NS	String	Sign string	15, 45
W	String	Temporary string	22, 30, 40
W	String	Temporary string	22, 30, 40, 45

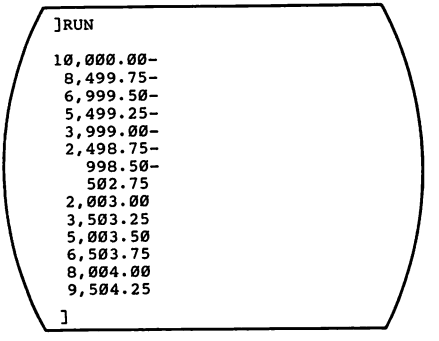


Fig. 4-5. Sample video screen.

largest number that we can format with this routine is 99999.99. Anything with a higher absolute value will be truncated on the left, with only the nine right digits showing. If you anticipate using larger numbers, then all that needs to be done is to increase the string limiter in line 45 from a nine to a larger digit.

Notice also in the above program that we have changed lines 100 through 130 to use larger numbers, and some negatives, so that the full formatting potential of the routine will be shown. When the program is run, the output should look like that in Fig. 4-5. In the next chapter, we shall use this technique to build output for formatted report lines.

CHAPTER 5

Report Formatting

Report formatting may be much more important to the success of a program than many people believe. Usually, the first contact that a person has with a program is the written output that it generates. The attractiveness and readability of that report may color the feelings that a person develops about your program.

If the report that a program generates is either too short or too long, it has failed the user. The report should adequately cover the informational needs of the user without burying him in a mountain of useless numbers or paper.

(In my view, programmers have a moral obligation to make computers useful, not to confuse the consuming public with meaningless numbers, figures, statistics, graphs, and reports that help nothing except the programmer's paycheck. The only reports that a program should generate are those that the user really needs and wants. To really impress a user with the number of reports your program will generate, allow him to select the individual reports he wants produced each time the program is run.)

Many times the word "report" conjures up visions of a stack of paper or fancy printers disgorging numbers at an ever increasing rate. Reports, however, are not limited to hardcopy (paper) output. They can be printed either to the screen or to the printer. It is especially desirable for a program to handle both types of output. This shows versatility, forethought, and planning on the part of the programmer. There should be separate sections of the program for both screen and printer output. This is because the screen reports must be formatted for only 40-column lines, while the printer reports generally need to be formatted for 80- or 132-column lines.

STANDARDIZATION

Reports generally consist of two parts. There are the HEADINGS, which usually appear only once on each page, and there are the INDIVIDUAL LINES that make up the body of the report. We will cover each area in detail.

When reading computer output, either video or printed, it is more pleasing to read output that has been "standardized". This means that all associated output begins in a certain column, or that the output is either left or right justified. Justification is the process used to make text either begin at the left margin (left justification), or end at the right margin (right justification), or both (fill justification). It is beyond the scope of this book to discuss fill justification, so we will stick with the first two in our discussion of standardization.

The first item of business, and one that should be covered when planning the program (you *do* plan your programs, don't you?), is to decide how the format of your output should be handled. This is the essence of standardization. For instance, when planning a report, there are many questions, such as these, that need to be answered:

1. What should appear on the heading?
2. Should the heading be centered?
3. Should a date and page numbers be included?
4. What size paper will this be printed on?
5. How wide is your report going to be?
6. How many lines for the heading?
7. How many columns?
8. What should the columns contain?
9. What order should the data be presented in?
10. What should the column headings say?
11. Should the report be single- or double-spaced?
12. How many blank lines should be left at the top of the screen?
13. How many blank lines should be left at the bottom of the screen?

It is best to get these questions out of the way in the planning stage of the program. The answers that you give may vary considerably from report to report, depending on factors such as the complexity of the report subject, value of the report, and intended report audience.

Since the answers to these questions differ according to individual needs, we will consider only the two basic components of a report—the headings and the lines that make up the body.

REPORT HEADINGS

Report headings are those lines which identify the report. The heading should contain items that would be of value in helping the

reader understand the contents of the report. Generally, a heading contains several standard items: a title, date, page number, column headings, and a divider. For instance, a sample report heading for a 132-column report appears in Fig. 5-1. Notice that each part of the heading is distinct and would be easily recognized by the reader.

Placement of all of these items in a heading depends, to a large degree, on how wide the report is to be. On narrow reports, you may want your heading to be several lines deep. On a wide report, the heading may be only one line. Most headings take a total (counting blank lines) of about 5 to 6 lines, but this may vary under any given conditions. For instance, notice that the sample report heading given in Fig. 5-2 is the same one referred to earlier, but it has been formatted for only an 80-column report.

The actual printing of the heading should be handled by a separate subroutine so that it may be called easily at the beginning of each page of output. A typical heading subroutine might look like the one in Listing 5-1 and use the variables in Tables 5-1 and 5-2.

This routine does several important things. First, it increments the page counter in line 50. This is so that you get a consecutive page number on each page. At some point before entering this heading routine for the first time, you will need to make sure that the variable P gets set to zero. Then your first page will always be 1. Generally, it is a good idea to begin your reports with the first page so that the reader doesn't think that you are only giving him a partial report.

The next item to be handled is the actual report title. Notice that this is centered for 80 columns. The centering is handled by the command POKE 36, 27. Since memory location 36 controls at which horizontal column the cursor is located, this command automatically moves the cursor to the 27th output column. This will always work with Apple 40-column video, and with most printers and interface cards. However, you may run into a problem with some 80-column cards, and with certain printer interface cards. If you do, then this should be replaced with a command similar to PRINT SPC$(27);, which will print 27 spaces. The decision on how to handle the tabbing is yours, and it should be based, to a large extent, on the intended hardware configuration. For the balance of our examples here, we will use the POKE 36, 27 technique.

It's also important to center the report title. This gives it added importance, and draws the reader's eyes right to the top middle of the page. After all, the title of the report is too important to be put where the reader must search for it. Here we have centered the title, as well as given it a full line.

The next line contains only two items, but for most reports, they are only slightly less important than the title. The date appears to the left side of the line, and the page number appears to the right. The page

```
                                        GOLIATH WIDGET CORPORATION
Week Ending 02/31/1984                                  Prepared by

                                                  ----------------------
Line  Metropolitan Location              Date      10000     11000
-------------------------------------------------------------------------
```

Fig. 5-1. Sample 132-column

```
                GOLIATH WIDGET CORPORATION - WEEKLY SUMMARY RECAP
Week Ending 02/31/1984                                          Page 1
-------------------------------------------------------------------------
Line   Location          Date      Account       Amount      Comments
-------------------------------------------------------------------------
```

Fig. 5-2. Sample 80-column report heading.

```
50   P = P + 1:REM   INCREMENT PAGE COUNTER
60   POKE 36,27:PRINT "SAMPLE TECHNICAL REPORT"
70   PRINT DAT E$;:POKE 36,71:PRINT "PAGE ";P
80   PRINT :PRINT HD$:REM  COLUMN HEADINGS
90   FOR J = 1 TO 80:PRINT "=";:NEXT :PRINT :PRINT
100  LC = 6:REM   SET LINE COUNTER TO INITIAL VALUE
110  RETURN
```

Listing 5-1

Table 5-1. Entry and Exit Variables for Report Headings Routine

On Entry:	On Exit
DATE$—Date for report HD$—Column headings P—Last page #	LC—Number of lines printed P—New page value

Table 5-2. Variable Table for Report Headings Routine

Variable	Type	Purpose	Used in Lines
DATE	String	Date	70
HD	String	Column headings	80
J	Numeric	Loop counter	90
LC	Numeric	Line counter	100
P	Numeric	Page number	50, 70

```
--------WEEKLY SUMMARY RECAPITULATION
       Johnathan Doe                                          Page 1

------------------ Accounts ----------------------------
       12000    17500    20100    23000    24000    25100    Comments
-------------------------------------------------------------------------
```

report heading.

number itself is set each time the heading routine is accessed, but the date must be set before entering the routine.

Next comes a blank line and the column headers. Here the variable HD$ was used for the headers line, but this could just as easily have been changed to the actual individual print commands. Generally, I like to use the string method because it is easier to change the headings that way, and it makes the routine look less cluttered.

Before entering this routine, HD$ needs to be set equal to the actual column headings that you want printed. All you need to do is grab your pencil and paper, and write out the column headings you want. Be sure to space between columns correctly when typing them into the machine. Chances are, your column heading line will need to be changed later, so don't throw your notes away yet.

Finally, there is a dividing line of equal signs, and then a blank line. These serve to separate the heading from the rest of the report.

The last part of the subroutine sets the line counter to six. If you will notice, this is the number of individual lines that have been printed by this routine. What this counter gets set to will vary (naturally) with the number of lines actually printed by your heading subroutine. The importance of this in the overall report will be discussed shortly.

As you have probably figured out by now, none of these routines are cast in concrete; they should be changed to reflect your specific needs at the time you are developing your program. What's important is that you develop a routine that will print a clear, concise heading for your reports.

After working out your heading routine in theory, it is generally best to make a few test runs to print your heading. Then you can determine if it will fit your needs, and make changes before you are too far into the actual report development. After all, if something works the first time around, it generally means that we have overlooked an obvious flaw which will make the program bomb later. This is just an irrefutable law of nature.

When your report headings are completed to your temporary satisfaction, then it is time to proceed to the body of the report.

```
10   HD$ = "EMPLOYEE NAME                    HOURS          RATE
          TOTAL PAY"

90    GOTO 1000

99   REM   NUMERIC INPUT ROUTINE

100  A$ = "":B$ = "":C$ = "":B = 0:D = 0

105  IF C$ = "." THEN D = 0:C$ = ""

110  GET A$:A$ = LEFT$ (A$,1)

115  IF A$ = CHR$ (13) THEN 150

116  IF A$ = CHR$ (8) AND LEN (B$) > 0 THEN C$ = RIGHT$ (B$,1):B$
      = LEFT$ (B$,LEN (B$) - 1):PRINT A$;" ";A$

117  IF A$ = CHR$ (8) THEN 105

118  IF B$ = "" AND A$ = "-" THEN 130

120  IF (A$ < "0" OR A$ > "9") AND A$ <  > "." THEN 110

125  IF A$ = "." AND D = 1 THEN 110

130  PRINT A$;

135  IF A$ = "." THEN D = 1

140  B$ = B$ + A$: GOTO 110

150  B = VAL (B$):RETURN

199  REM   ALPHANUMERIC INPUT ROUTINE

200  A$ = "":B$ = ""

210  GET A$:A$ = LEFT$ (A$,1):A = ASC (A$)

220  IF A = 13 THEN PRINT :RETURN

230  IF A = 8 AND LEN (B$) > 0 THEN B$ = LEFT$ (B$,LEN (B$ - 1)):
     PRINT A$;" ";A$;

240  IF A = 8 THEN 210

250  IF A < 32 OR A > 90 THEN 210

260  PRINT A$;

270  B$ = B$ + A$: GOTO 210

299  REM   ROUNDING ROUTINE

300  N = INT (N / .01 + .5) * .01

310  RETURN

399  REM   ROUTINE TO CONVERT NUMBERS FOR OUTPUT

610  POKE 36,30:N = H:GOSUB 400:PRINT A$;
```

Listing 5-2

```
400  N = INT (N / .01 + .5) * .01

410  NS$ = " ":IF N < 0 THEN NS$ = "-":N = ABS (N)

420  N1 = INT (N):N2 = N - N1

430  W$ = STR$ (N1):W1$ = "":IF N1 = 0 THEN W$ = "0"

440  D$ = STR$ (N2):IF N2 = 0 THEN D$ = ".00"

450  IF LEN (D$) < 3 THEN D$ = D$ + "0"

460  IF LEN (W$) > 3 THEN W1$ = "," + RIGHT$ (W$,3) + W1$:W$ = LE
     FT$ (W$,LEN (W$) - 3): GOTO 460

470  W1$ = W$ + W1$

480  A$ = RIGHT$ ("           " + W1$ + D$,9) + NS$

490  RETURN

499  REM   ROUTINE TO PRINT REPORT HEADINGS

500  P = P + 1:REM   INCREMENT PAGE COUNTER

510  POKE 36,27:PRINT "PAYROLL WORKSHEET REPORT"

520  PRINT DA$;:POKE 36,71:PRINT "PAGE ";P

530  PRINT :PRINT HD$:REM  COLUMN HEADINGS

540  FOR J = 1 TO 80:PRINT "=";:NEXT :PRINT :PRINT

550  LC = 6:REM   SET LINE COUNTER TO INITIAL VALUE

560  RETURN

599  REM   ROUTINE TO PRINT INDIVIDUAL REPORT LINE

600  PRINT CHR$ (4)"PR#1":PRINT N$;

620  POKE 36,35:N = R:GOSUB 400:PRINT A$;

630  POKE 36,50:N = PA:GOSUB 400:PRINT A$

640  PRINT CHR$ (4)"PR#0":RETURN

999  REM   MAIN CONTROL ROUTINES

1000 P = 0:DA$ = "01/01/99":PRINT CHR$ (4)"PR#1":GOSUB 500:PRINT
     CHR$ (4)"PR#0"

1010 PRINT "EMPLOYEE NAME: ";:UL = 20:GOSUB 200:N$ = B$

1015 IF B$ = "" THEN 1050

1020 PRINT "HOW MANY HOURS: ";:GOSUB 100:H = B

1030 INPUT "WHAT RATE OF PAY: ";:GOSUB 100:R = B

1040 PA = H * R:GOSUB 600:PRINT : GOTO 1010

1050 PRINT CHR$ (4)"PR#1":PRINT CHR$ (12):PRINT CHR$ (4)"PR#0":EN
     D
```

Listing 5-2 cont.

Table 5-3. Variable Table for Payroll Program

Variable	Type	Purpose	Used in Lines
A	String	Keypress and formatted output	100, 110, 115, 116, 117, 118, 120, 125, 130, 135, 140, 200, 210, 230, 260, 270, 480, 610, 620, 630
A	Numeric	ASCII value	210, 220, 230, 240, 250
B	String	Input string	100, 116, 118, 140, 150, 200, 230, 270, 1010, 1015
B	Numeric	Numeric value	100, 150, 1020, 1030
C	String	Backspace value	100, 105, 116
D	Numeric	Flag	100, 105, 125, 135
D	String	Temporary string	440, 450, 480
DATE	String	Date	520, 1000
H	Numeric	Hours	610, 1020, 1040
HD	String	Column headings	10, 530
J	Numeric	Loop counter	540
LC	Numeric	Line counter	550, 640
N	Numeric	Number to be rounded and formatted	300, 400, 410, 420, 610, 620, 630
N	String	Name	600, 1010
N1	Numeric	Integer value	420, 430
N2	Numeric	Integer value	420, 440
NS	String	Sign string	410, 480
P	Numeric	Page number	500, 520, 1000
PA	Numeric	Total pay	630, 1040
R	Numeric	Pay rate	620, 1030, 1040
UL	Numeric	Upper limit	1010
W	String	Temporary string	430, 460, 470
W1	String	Temporary string	430, 460, 470, 480

LINE PREPARATION

The body of any report is usually made up of individual data lines which are repeated over and over again, until the appropriate number of lines have been printed on the page.

Preparing those lines for output is simply a matter of collecting the data you wish to print, then printing it at the right place on the paper. To illustrate this, we need to pull together a program which includes all of the subroutines that we have developed up to this point. This program will allow a user to input individual payroll information, and then print out the resultant information in an 80-column format. This will *not* be a marketable (or even useful) payroll program. However, it will illustrate the points that need to be made for our demonstration. The program would look like the one in Listing 5-2, for which variables are defined in Table 5-3.

Now that we have put together many of the routines from earlier chapters, our program gets quite a bit longer. If you were to run this, you might be surprised to find that it does quite a nice job of printing a usable report and even handles multiple pages. Most of the routines included in this program are presented elsewhere in this book, so we will only concentrate on lines 600 through 699. These lines print the individual lines that make up the body of the report.

Line 600 directs all subsequent output to the printer, and then prints the employee's name. In line 610, we position the cursor to print the next variable. By use of the POKE 36, 35 command, we will start printing our next output in the 35th column. Then we convert this output to a formatted string and print it.

The balance of the routine enacts this same process for both the rate and the total pay. Then, in line 640, we increment the line counter (L). If it is above a certain number of lines (here we use 60) then we do a form feed and print another heading for the next page.

Finally, the output is directed back to the screen, and we return from our routine.

This routine is pretty efficient, and works well for this report. Naturally, the routine may become far more complex for other reports, depending on the complexity of the data to be printed. Another change that may be desirable for different reports is a line that will keep a running total of the amounts printed. Then, when the printing of data is complete, you can print the totals for the report.

SPECIAL LINE POSITIONING

There is one last area that should be covered concerning reports. When printing information, it may be desirable to left justify, center, or right justify your output.

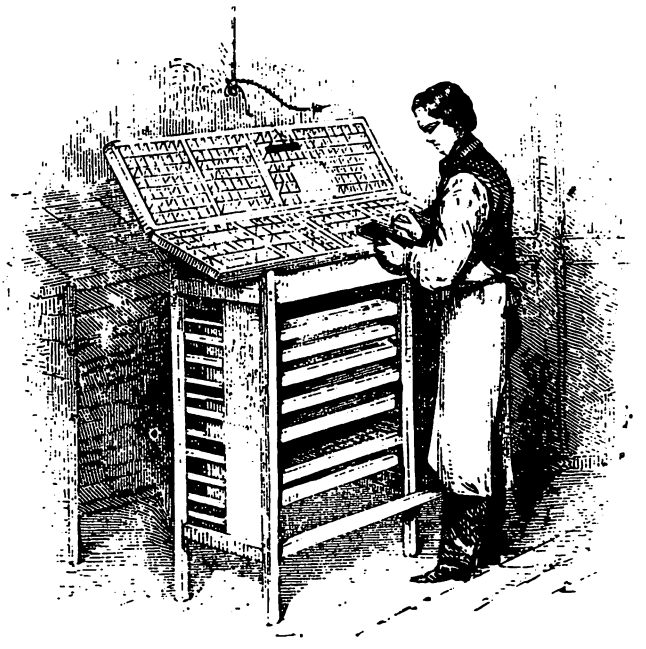


Fig. 5-3. Early manual line formatter.

Left justification is simple. All you need to do is position your cursor to the column at which you wish to begin printing, then print it. Simple, huh?

Centering is a little more tricky, but if handled in a common subroutine, it can be made simpler. Once again make sure that the common variable (A$) is equal to the string that is to be centered. The routine looks like this:

```
100 POKE 36, 20 - LEN(A$) / 2: PRINT A$: RETURN
```

The heart of the routine is the formula 20 − LEN (A$) / 2. Simply, this routine takes half the length of the string to be centered, subtracts it from the center point of the output line, and then moves the cursor to that column. Then the string is printed.

This is a centering routine for a 40-column output (screen or printer). In order to make it work for any other size output line, change the 20 to a number that represents half the length of your output line. For example, use 40 for 80-column centering, 36 for 72-column centering, 66 for 132-column centering, etc.

In case you were wondering, this routine will not work if you try to center a string that is longer than the line you want to center it on.

You can't POKE a negative value into a memory location. If you try, an ?ILLEGAL QUANTITY ERROR will be generated. This will also happen if the value you try to POKE works out to be greater than 255, which should not happen anyway. It's doubtful you will be centering on a line longer than 510 characters—few printers will handle paper that wide.

Right justification is no harder than centering as long as you approach it in the same manner. All that right justification means is that you want each string printed so that it ends at a specific column. Here is a routine that will handle this:

```
100 POKE 36, 39 - LEN(A$): PRINT A$: RETURN
```

Simple, right? Right! All that needs to be done is to subtract the length of the string to be printed from a number that represents the column at which you wish the string to end.

I suppose it goes without saying that if you want the printing to end at a different place from column 39, all you need to do is change the number 39 to a different one that represents the rightmost printing position. But I'll remind you anyway.

Now you know all of the deep, dark trade secrets of printing reports and using strings to get the output appearance that you desire. All you need to do now is arrange them in the order necessary to get the report that you want. The creative process that you go through to do this may be the toughest part of programming.

In other chapters in this book, routines are discussed that may be helpful to you when formatting output for your report.

CHAPTER 6

Working with Dates

Many programmers tend to consider themselves mental and/or social deviates. Before those of you who fit this mold get the wrong idea about this chapter, let me tell you what kind of dates we will be working with. We will only be working with the kind that determine our present position in relation to the continuum that stretches from yesterday through tomorrow. Any other dates are outside the scope of this chapter, and indeed this book.

Many programs work with dates, either inputting, manipulating, or printing them. This chapter will discuss how to standardize dates, how to let the user input dates, how to manipulate them, and how to print them. First, however, we should give a little general purpose information concerning dates.

DATES, CALENDARS, AND DATING CONVENTIONS

Since the beginning of time, man has kept track of dates according to many different methods. The sun and moon are two of the most popular measuring devices for time. Our present calendar is based on the sun, so we'll discuss that now.

Our present calendar is called the Gregorian calendar, after Pope Gregory XIII. It was adopted by most of the world in 1582 to correct errors in the previous (Julian) calendar. The transition between the two was rather abrupt. Ten days were dropped from the middle of October to correct differences between the sun and the calendar, since that was easier than moving the earth or sun to match the calendar. This procedure corrected the next equinox to its proper date.

So, October 5, 1582 became October 15, 1582, and most of the world started on the new calendar system. (Most, that is, except for certain Germanic nations that adopted it about 1700. Great Britain

did not change until 1752, Russia until 1918, and Turkey was the last nation to switch in 1927.)

The Gregorian calendar is based largely on the Julian calendar with 12 months of varying length; however, the new calendar corrected cumulative errors in the Julian calendar by changing the method of figuring leap year days. Instead of a leap year every 4 years, the Gregorian calendar allowed for a leap year every 4 years except in century years. In these years, there would be no leap year unless the year was divisible by 400. So, for example, 1900 was not a leap year, since it is not divisible by 400. However, 2000 will be a leap year for exactly the same reason.

A lot of the problems with checking dates and comparing them has to do with leap years, as will be noticed later. First, however, we should discuss standardization of dates.

DATE STANDARDIZATION

If you haven't noticed, everyone and his brother uses a different notation for dates. For instance, each of the following represents the same date:

August 18, 1983
18 August 1983
8/18/83
08/18/83
18/8/83
18/08/83

To complicate matters further, the year portion of each date could have used four digits (1983) instead of two (83), or each date could have used a different field separator, such as a dash (-) instead of the slash (/). If you allow the users of your program to input dates in any format they desire, you will usually get one more format than you were able to think of. This, in turn, will cause your program to go bonkers.

I rest the case for date standardization. It should be obvious that if you are going to do any type of manipulation with dates, you need to have some sort of standard that will be followed by the users.

The best way to proceed is to determine, if possible, the standard for other programs that the user may be running. It would be frustrating to have to input dates differently for each program. For a user, entering information should be as painless as possible, particularly entering dates. Furthermore, if the user is familiar with one type of notation, he will loathe making any change just because you happen to like it a different way. If you can, determine what the user wants and is used to doing.

If you are fortunate enough to have an area where there is no standard for inputting dates, then it is up to you to determine one. Remember, however, that your preference may not be the market preference. Discretion is always advised.

You will find that most computer programs use the MM/DD/YY format for inputting dates. Computers work better with numbers than with names of months. Besides, how many corporate executives know how to spell "February" correctly? Allowing the user to spell out the name of the month will usually open the door to frustration because of possible misspellings.

INPUTTING DATES

Now that you have decided on a standard, let's look at a method for inputting dates. Since the MM/DD/YY format is the most common, we will use that in our discussions and examples. If you decide to use some other format, you will definitely have to change the routine to reflect that.

Basically, all we have to do is allow the user to input a string of characters, tear them apart to check their validity, and then put them back together in a standardized format.

The prompt that you present to the user should also show what format the program expects the response to follow. For instance, presenting the user with **DATE (MM/DD/YY):** leaves little room for doubt as to what is expected. The actual input routine then used should be rather selective. Chapter 2 of this book presents a nifty string input routine that will work just fine.

After getting the input from the user, the next step is to tear it apart to check its validity. For instance, someone may input the date as September 31. Since we all know that this date only rolls around once every 327 years, we have to gently remind the user to re-enter the date again. The following routine will tear the string apart to check for the validity of a date. It is then formatted to the standard (MM/DD/YY), and returned to the caller. On entry, A$ should contain the string to be analyzed. On its successful exit, MM contains the number of the month (1-12), DD contains the day (1-valid for month), and YY contains the year. The re-formatted date is also back into A$. This program is shown in Listing 6-1 with variables summarized in Tables 6-1 and 6-2.

Notice several things about this routine. First of all, it checks the length of A$ upon entry. If it is less than six characters, or greater than eight characters, then the user has apparently made an invalid date entry. The smallest possible date in the MM/DD/YY format would be something like "1/1/83", which is only six characters. If the month and day were two characters each, then the string would still only be 8 characters long. Anything longer doesn't fit the format.

```
100  MM = 0:DD = 0:YY = 0

110  K = LEN (A$):IF K < 6 OR K > 8 THEN 210

120  MM = VAL (A$):FOR J = 1 TO K

130  IF VAL (MID$ (A$,J,1)) = 0 AND MID$ (A$,J,1) < >
     "0" THEN DD = YY:YY = J + 1

140  NEXT :IF DD = 0 THEN 210

150  DD = VAL (MID$ (A$,DD,2)):YY = VAL (MID$ (A$,YY))

160  IF MM < 1 OR MM > 12 THEN 210

170  IF DD < 1 OR DD > 31 THEN 210

180  IF MM = 2 AND (INT (YY / 4) * 4 < > YY) AND (INT
     (YY / 400) * 400 < > YY) AND DD > 28 THEN 210

190  IF (MM = 4 OR MM = 6 OR MM = 9 OR MM = 11) AND DD
     > 30 THEN 210

200  A$ = RIGHT$ ("00" + STR$ (MM),2) + "/" + RIGHT$ ("
     00" + STR$ (DD),2) + "/" + RIGHT$ ("00" + STR$ (YY
     ),2):RETURN

210  A$ = "":RETURN
```

Listing 6-1

Table 6-1. Entry and Exit Variables for Date Analysis Routine

On Entry:	On Exit:
A$—Preliminary date	A$—Formatted date
	DD—Day
	MM—Month
	YY—Year

If the date falls outside of these length specifications, then execution branches to line 210. All of our early exits from this routine are by way of line 210. You can test for a correct completion of this routine by checking to see if A$ is set to anything. If it is not, you should have the program loop back to get the date again.

The next several instructions are used to tear A$ apart so it can be completely analyzed. Line 120 sets MO (month variable) to the value of A$. The VAL function will only convert up to the first non-numeric character, so it will only convert those numbers to the left of the first slash or dash.

Next, lines 120 through 140 loop through the entire A$ to find the delimiters. A delimiter is a non-numeric divider between the components of the date. In the string "01/02/84", the slashes are delimiters.

Table 6-2. Variable Table for Date Analysis Routine

Variable	Type	Purpose	Used in Lines
A	String	Date (in & out)	110, 120, 130, 150, 200, 210
DD	Numeric	Day	100, 130, 140, 150, 170, 180, 190, 200
J	Numeric	Loop counter	120, 130
K	Numeric	Length of input	110, 120
MM	Numeric	Month	100, 120, 160, 180, 190, 200
YY	Numeric	Year	100, 130, 150, 180, 200

This routine finds those. Then DD (day variable) is assigned to be equal to YY (year variable), and YY is set equal to the character position following the position of the delimiter within the string.

This process is repeated until the end of the string is reached. If DD is still equal to 0 upon exit from this loop, then we know that the user did not enter an entire date. If this is the case, the routine is exited early by way of line 210.

Line 150 sets DD and YY equal to the value of the two consecutive characters following the delimiters. Remember that DD and YY were set equal to these positions in the previous analysis loop.

Line 160 checks to make sure that the month is a valid number. If it is too small or large, then a branch is made to line 210 for an early routine exit. Lines 170 through 190 then make sure that the day entered is valid for the month that was entered. Line 180 even allows for leap years based on the year entered. As you will remember from our earlier discussion, we have to check both for years divisible by 4 and centuries divisible by 400 to be accurate in our date figuring. If an illegal date is detected—as in our September 31 example, where the day was invalid for the month—then the routine jumps to line 210.

Finally, line 200 reassembles A$ to exactly 8 characters by padding each component of the date with a leading zero if needed. It also uses dashes as delimiters, but this could easily be changed to some other character if desired. When this is done, execution is returned to the caller.

By using this routine, several things are accomplished. First of all, you an find Month, Day, and Year values for future use in calculations. Secondly, each date will be formatted in exactly the same way. In fact, it is possible for someone to enter a date as "6:1*84", and

have it return as "06/01/84". This standardization is imperative for orderly conduct of a program.

There are a few conditions where this routine will not work. For instance, if someone enters a date such as "6 1 83", then the month will be translated in line 120 as 6183. This will cause the program to exit early from line 160. This should be the only condition that may cause confusion, however. If you check for the length of A$ after calling this routine, then you can always go back and ask for the date again.

MANIPULATING DATES

In order to manipulate a date, it must be converted to a single number for comparison, addition, or whatever. This would be rather simple to do, if it weren't for leap years. However, the following formula will accomplish this task quickly:

```
10 DEF FN DA(X) = 365 * YY + DD + 31 * (MM - 1)
   - ((MM > 2) * ( INT (.4 * MM + 2.3))) +
   INT ((YY - (MM < 3)) / 4) -
   INT (.75 * ( INT ((YY - (MM < 3)) /
   100 + 1)))
```

Pretty long equation, right? We thought it best to keep it as one equation, and to make it a function so that it could be utilized anywhere within a program. To use it, all you need to use is **N= FN DA(X)**, and N will be made equal to the unique value corresponding to the date entered.

When using this statement, it is assumed that MM, DD, and YY have already been set to valid (checked) values. YY can either be set to a two-digit value, or a four-digit value. If you are sure that all dates entered will be within the same century, then two-digit values for years would be fine; however, if this is not the case, then four-digit values will be required. In any case, the year should never be less than 1582, because that is the year the Gregorian calendar was adopted, and this formula is based on the Gregorian calendar.

This formula converts any given date to a number equivalent to the total days represented by the date. In order to understand this formula completely, let's look at each part of it individually, starting with the formula portion of the statement itself. It is assumed that you already know the purpose of defining functions and how to utilize them.

```
10 DEF FN DA(X) = 365 * YY + DD + 31 * (MM - 1)
   - ((MM > 2) * ( INT (.4 * MM + 2.3))) +
   INT ((YY - (MM < 3)) / 4) -
   INT (.75 * ( INT ((YY - (MM < 3)) /
   100 + 1)))
```

This portion of the formula converts raw years to days, and adds the partial portion of the month to the number. Then the number of days in the completed months of the year is added to that number.

```
10 DEF FN DA(X) = 365 * YY + DD + 31 * (MM -1)
   - ((MM > 2) * ( INT (.4 *MM + 2.3))) +
   INT ((YY - (MM < 3)) / 4) -
   INT (.75 * ( INT ((YY - (MM < 3)) /
   100 + 1)))
```

In the previous part of the formula, we based all months on 31 days. This part of the formula corrects for that by subtracting the correct number of days from our number. This, however, should only happen if the date is later than or equal to March 1st, so we do a Boolean check during this portion of the equation. If it is false, then this whole part of the equation will be equal to zero. Only if it is true (that MM will be greater than 2) will this portion of this formula be utilized.

```
10 DEF FN DA(X) = 365 * YY + DD + 31 * (MM - 1)
   - ((MM> 2) * ( INT (.4 * MM + 2.3))) +
   INT ((YY - MM < 3)) / 4)-
   INT (.75 * ( INT ((YY - (MM < 3)) /
   100 + 1)))
```

This part of the equation adds the number of leap days that have occurred so far. If the date being analyzed is in January or February, YY is decremented by one so that we are not counting a leap day that hasn't happened yet.

```
10 DEF FN DA(X) = 365 * YY + DD + 31 * (MM - 1)
   = ((MM > 2) * ( INT (.4 * MM + 2.3))) +
   INT ((YY - (MM < 3)) / 4) -
   INT (.75 * ( INT ((YY - (MM < 3)) /
   100 + 1)))
```

This final part of the formula corrects for the century years that do not contain leap days. It also takes into account that we may be analyzing a January or February date in a century year by decrementing YY if this is the case.

This formula is accurate, as long as certain conditions are met. First of all, you should make sure that MM, DD, and YY are *valid figures*. If they are derived after going through the date input routine discussed earlier in this chapter, then they should be correct.

Another consideration is that the year variable (YY) should never be equal to zero. So that this doesn't happen, it is a good idea to use four-digit years at all times. This is not necessary, but advisable. This way you will be sure of unique numbers. This routine is only accurate for dates after October 15, 1582. Usually, however, you will not have

someone entering a date earlier than that. Remember that there were no actual dates between 10/5/1582 and 10/14/1582. These days were dropped when the new Gregorian calendar was instituted.

Some examples of the unique numbers associated with certain dates are shown in Table 6-3. These numbers were derived by using this formula.

Once this formula has been executed, the resultant number can be stored so that it can later be compared to resultant numbers from other dates. In this way, you can get an idea of how many days between dates, or which date came first, etc.

Table 6-3. Unique Numbers for Specific Dates

Date	Unique #
06/11/1956	714576
05/25/1983	724420
02/29/1984	724700
03/01/1984	724701
11/05/1984	724950
12/31/1999	730484
01/01/2000	730485
08/12/2013	735457

DAYS OF THE WEEK

Once you have performed the above function on a date, you can do one more function to determine the day of the week based on what date was entered. The series of lines in Listing 6-2 will do it for you. It uses the variables summarized in Tables 6-4 and 6-5.

This routine basically finds the remainder after dividing evenly by 7. It is then rounded to return a number between 0 and 6. These represent the days of the week between Saturday and Friday.

Pretty simple, right? But how could this be used in a program? In a business program it would be useful to have a user enter today's date when he starts using the program. Then you would be able to return a standardized date and even the day of the week. Beyond that, these routines are great for figuring true aging, and if you look around, there are bound to be numerous other uses as well.

PRINTING DATES

There are many ways to print dates. The best way is the way that will be of the most use to your intended audience.

For most people the format "MM/DD/YY" will be sufficient. However, you may wish to use the actual name of the month in the format

```
50   W$(Ø) = "Saturday":W$(1) = "Sunday":W$(2) = "Monda
     y":W$(3) = "Tuesday":W$(4) = "Wednesday":W$(5) = "
     Thursday":W$(6) = "Friday"

60   D = INT (7 * (N / 7 - INT (N / 7)) + .5)

70   PRINT W$(D)
```

Listing 6-2

Table 6-4. Entry and Exit Variables for Day of Week Routine

On Entry:	On Exit
N—Unique calculated date value	D—Numeric day of week W$(*)—Days of week

Table 6-5. Variable Table for Day of Week Routine

Variable	Type	Purpose	Used in Lines
D	Numeric	Day of week	60, 70
N	Numeric	Unique date	60
W$(*)	String	Days of week	50, 70

"September 22, 1983". This is easy if you have analyzed the date previously. All you need to do is have an alphanumeric array set up with the names of the months of the year in a 1 through 12 matrix. Then print the name of the month using MM as a subscript. This is basically the same thing that was done with the day of the week.

The most important consideration when printing dates is to be consistent. Consistency, in any format, looks professional; inconsistency will be noticed immediately.

PULLING IT ALL TOGETHER

So far, we've covered a lot of ground on dates. If we put these routines all together, we can create the useful (albeit rather trivial) program shown in Listing 6-3, with variables defined in Table 6-6.

This program requests two dates from a user. First it requests today's date, and finally the user's birthdate in lines 1010 and 1040. The program analyzes each date as it is entered (lines 100 through 210) in order to determine if it is in the correct format. If not, then a bell is sounded and the questions are asked again (lines 1020 and 1050).

Finally, the dates are compared (lines 1030 and 1060) and the days

```
10   DEF FN DA(X) = 365 * YY + DD + 31 * (MM - 1) - ((MM > 2) * (
     INT (.4 * MM + 2.3))) + INT ((YY - (MM < 3)) / 4) - INT (.75
      * (INT ((YY - (MM < 3)) / 100 + 1)))

20   W$(0) = "Saturday":W$(1) = "Sunday":W$(2) = "Monday":W$(3) =
      "Tuesday":W$(4) = "Wednesday":W$(5) = "Thursday":W$(6) = "F
     riday"

30    GOTO 1000

100  MM = 0:DD = 0:YY = 0

110  K = LEN (A$):IF K < 8 OR K > 10 THEN 210

120  MM = VAL (A$):FOR J = 1 TO K

130  IF VAL (MID$ (A$,J,1)) = 0 AND MID$ (A$,J,1) <  > "0" THEN D
     D = YY:YY = J + 1

140  NEXT :IF DD = 0 THEN 210

150  DD = VAL (MID$ (A$,DD,2)):YY = VAL (MID$ (A$,YY))

160  IF MM < 1 OR MM > 12 THEN 210

170  IF DD < 1 OR DD > 31 THEN 210

180  IF MM = 2 AND (INT (YY / 4) * 4 <  > YY) AND (INT (YY / 400)
      * 400 <  > YY) AND DD > 28 THEN 210

190  IF (MM = 4 OR MM = 6 OR MM = 9 OR MM = 11) AND DD > 30 THEN
     210

200  A$ = RIGHT$ ("00" + STR$ (MM),2) + "/" + RIGHT$ ("00" + STR$
      (DD),2) + "/" + RIGHT$ ("00" + STR$ (YY),2):RETURN

210  A$ = "":RETURN

300  A$ = "":B$ = ""

310  GET A$:A$ = LEFT$ (A$,1):A = ASC (A$)

320  IF A = 13 THEN PRINT :RETURN

330  IF A = 8 AND LEN (B$) > 0 THEN B$ = LEFT$ (B$,LEN (B$) - 1):
     PRINT A$;" ";A$;

340  IF A = 8 THEN 310

350  IF A < 32 OR A > 90 THEN 310

360  PRINT A$;

370  B$ = B$ + A$: GOTO 310

1000 TEXT :HOME

1010 PRINT "ENTER TODAY'S DATE (MM/DD/YYYY): ";

1020 GOSUB 300:A$ = B$:GOSUB 100:IF A$ = "" THEN PRINT CHR$ (7);:
      GOTO 1010
```

Listing 6-3

```
1030 D1$ = A$:D1 = FN DA(X)

1040 PRINT :PRINT "ENTER YOUR BIRTHDAY (MM/DD/YYYY): ";

1050 GOSUB 300:A$ = B$:GOSUB 100:IF A$ = "" THEN PRINT CHR$ (7);:
     GOTO 1010

1060 D2$ = A$:D2 = FN DA(X)

1070 W1 = INT (7 * (D1 / 7 - INT (D1 / 7)) + .5)

1080 W2 = INT (7 * (D2 / 7 - INT (D2 / 7)) + .5)

1090 DIF  = ABS (D1 - D2)

1100 HOME :PRINT "TODAY IS ";W$(W1);", ";D1$

1110 PRINT "YOU WERE BORN ON ";W$(W2);", ";D2$

1120 PRINT "THAT WAS ";DIF ;" DAYS AGO."

1130 END
```

Listing 6-3 cont.

of the week that correspond to those dates are figured (lines 1070 and 1080). The results are then printed and the program ends.

This is a pretty good example of how to use the information contained in this chapter. Notice also that the input routine contained in lines 300 through 370 is the one that was developed in Chapter 2.

In the next chapter, time will be dissected within your computer.

Table 6-6. Variable Table for Birthday Program

Variable	Type	Purpose	Used in Lines
A	String	Input/Keypress String	110, 120, 130, 150, 200, 210, 300, 310, 330, 360, 370, 1020, 1030, 1050, 1060
A	Numeric	ASCII keypress	310, 320, 330, 340, 350
B	String	Input string	300, 330, 370, 1020, 1050
D1	String	Today's date	1030, 1100
D1	Numeric	Unique value of today's date	1030, 1070, 1090
D2	String	Birthday	1060, 1110
D2	Numeric	Unique value of birthday	1060, 1080, 1090
DA(*)	Numeric Function	Derive unique number from date	10, 1030, 1060
DD	Numeric	Day	10, 100, 130, 140, 150, 170, 180, 190, 200
DIF	Numeric	Date difference	1090, 1120
J	Numeric	Loop counter	120, 130
K	Numeric	Length of string	110, 120
MM	Numeric	Month	10, 100, 120, 160, 180, 190, 200
W(*)	String	Days of week	20, 1100, 1110
W1	Numeric	Day of week	1070, 1100
W2	Numeric	Day of week	1080, 1110
YY	Numeric	Year	10, 100, 130, 150, 180, 200

CHAPTER 7

Time and Time Again

Have you ever noticed that we have a strange way of telling time in our society? Time is based on sixties. There are sixty seconds in a minute, and sixty minutes in an hour. After that it really gets messed up. Well, a computer can handle all sorts of messy items and details, as long as it is programmed correctly to begin with.

In this chapter we will discuss all of the nitty-gritty details that have to do with hours, minutes, and seconds. We will not discuss how to time events, since software timers are generally very inaccurate, particularly over long periods of time. Our attention, instead, will be directed towards manipulating units of time—in other words, how to calculate the interval between a given beginning time and a given ending time.

INPUTTING TIME

For most applications which require time calculations, having a user input a beginning and an ending time by way of the keyboard will be sufficient. As with dates (see Chapter 6), the entry that the user makes will need to be analyzed to make sure that it is formatted correctly and that it is within bounds.

If more accurate (or trustworthy) input of time is needed, there are numerous peripherals available that provide a direct input of time to a program. These are usually called "chronographs" or "clock cards". If you determine that you need one, your local computer store should be able to help you pick one out. The use and interfacing of these devices is beyond the scope of this chapter. Because of the virtual multitude of actual peripherals that exist for the purpose of inputting time to a computer, describing them all here would be a monumental task.

Fig. 7-1. Father Time.

The time string can best be entered by using the normal string input routine (see Chapter 2). When it comes to analyzing it, however, a specialized routine will be needed. The complexity of those routines depends, in large part, on the accuracy to which you wish later calculations to be performed. For instance, if you only need accuracy to the nearest minute, then you need only expect a user to input time as hours and minutes. If, however, you need to account for seconds, then the user will need to enter those as well.

In this section we will give examples of routines that may be used for both these degrees of accuracy. First, let's look at Listing 7-1, a routine which could be used to analyze strings consisting of hours and minutes separated by a colon, in the format "HH:MM T". The variable "T" stands for either an 'A' or 'P', depending on whether or not the time being entered is 'AM' or 'PM'; other variables are summarized in Tables 7-1 and 7-2.

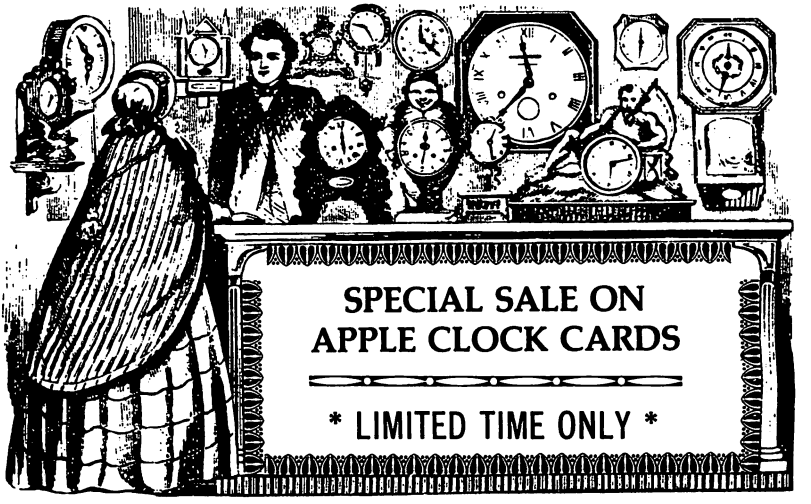


Fig. 7-2. Typical clock card store.

```
100  REM   ANALYZE HOURS AND MINUTES ONLY

105  IF LEN (A$) < 4 THEN 195

110  IF RIGHT$ (A$,1) <  > "A" AND RIGHT$ (A$,1) <  > "
     P" THEN 195

120  H = VAL (A$):M = 0:FOR J = 1 TO LEN (A$):IF VAL (M
     ID$ (A$,J,1)) = 0 AND MID$ (A$,J,1) <  > "0" THEN
     M = J + 1:J = LEN (A$)

130  NEXT :IF M = 0 THEN 195

140  M = VAL (MID$ (A$,M)):IF M < 0 OR M > 59 THEN 195

150  IF H < 1 OR H > 12 THEN 195

160  A$ = RIGHT$ ("00" + STR$ (H),2) + ":" + RIGHT$ ("0
     0" + STR$ (M),2) + " " + RIGHT$ (A$,1)

170  T = H * 60 + M:IF T > 719 THEN T = T - 720

180  IF RIGHT$ (A$,1) = "P" THEN T = T + 720

190  RETURN

195  A$ = "":RETURN
```

Listing 7-1

Table 7-1. Entry and Exit Variables for Time Routine (Hours and Minutes)

On Entry:	On Exit:
A$—String to be analyzed Format: HH:MM T HH—Hours MM—Minutes T—'A' or 'P'	A$—Return string Format: HH:MM T HH—Hours MM—Minutes T—'A' or 'P' If equal to " " (null) then error in input string. H—Hours M—Minutes T—Total Minutes

Table 7-2. Variable Table for Time Routine (Hours and Minutes)

Variable	Type	Purpose	Used in Lines
A	String	Entry/Exit	105, 110, 120, 140, 160, 180, 195
H	Numeric	Hours	120, 150, 160, 170
J	Numeric	Loop Counter	120
M	Numeric	Minutes	120, 130, 140, 160, 170
T	Numeric	Total Minutes	170, 180

This routine does several things. Before we explain what it does, however, let's look at the variables used in this routine. A$ is used both as input and as the primary return value. This is the string to be checked, and it is the completely formatted string that is returned when the routine is completed. The variables H, M, and T stand for Hours, Minutes and Total minutes, respectively. Total minutes is the total amount of time represented by the input string and calculated according to a 24-hour clock. Thus, T can vary from 0 to 1439 (number of minutes in a day, minus one).

Now for the routine itself. First, line 105 checks to make sure that the length of the string entered is acceptable. If it is not at least four characters in length, then execution is transferred to line 195 where a null string is returned to the caller.

Line 110 makes sure that the rightmost character of the string entered is either an "A" or a "P". This signifies either AM or PM, and it saves the user from manually calculating time on a 24-hour basis. If the input string does not end with either of these characters, then an early exit is taken through line 195.

Line 120 breaks down the string into its preliminary component parts. H is set equal to the value of the user's entry, which will return an actual value up to the first non-numeric character in the string. Then each character of the string is examined to see if it is a non-numeric character. This first non-numeric character is assumed to be the delimiter between hours and minutes. If such a delimiter is found, then M is set equal to the character position following the non-numeric character.

This loop is terminated in line 130. Upon termination, if M still equals zero, then an early exit is taken. This is because M will only be equal to zero if no delimiter was found in the preceding loop. Since this is an "illegal" format, an early exit is mandated.

Line 140 sets M equal to the minute value of the input string and checks it to make sure it is legal. If it isn't between 0 and 59, then an early exit is taken.

Line 150 does basically the same thing with hours. If the number of hours entered was less than one or greater than 12 then an error is indicated by exiting through line 195.

By the time the program executes line 160, the input string has been completely checked and found to be valid. This line puts the derived values back together in a standard format. Someone could have put in the time as "1/1P", and when this line is executed, the return string will be equal to "01:01 P". Notice that this re-formatted string is assigned to A$, the same variable that was previously the input string. This helps "conserve" variable space and frees you from having to re-define any variables after returning from this routine.

Line 170 and 180 calculate the total number of minutes signified by the time which was input. This figure is assigned to the variable T, and can be used in later calculations for elapsed *and cumulative* time.

After this routine is called and executed, the length of A$ should be checked. If the length is equal to zero (null), then an error was detected in the user's entry. At this point, the time entry should be requested again. Checking the length of A$ is the quickest way to determine the validity of the time entered by the user.

A final note is in order here. In analyzing a user's input, it is assumed that noon is entered as "12:00 P", and midnight as "12:00 A". Obviously, if these are confused by a user, then later calculations will be inaccurate.

HOURS, MINUTES AND SECONDS

The preceding routine, as stated earlier, is fine for figuring hours and minutes, and will be adequate for the majority of applications. A few people need to also figure seconds, though. With slight modifi-

cations, this routine can also serve their needs. Listing 7-2 shows the routine slightly modified to account for seconds. The variables are shown in Tables 7-3 and 7-4.

Since this routine is basically the same one we used to calculate hours and minutes, we will only examine those lines that were changed.

Line 105, which still checks the length of the user's entry, now allows for a minimum string length of six characters. Thus, "1:0:0A" is the minimum that could be entered.

The verification loop in lines 120 and 130 has been expanded to include the variable "S" which is used to denote seconds. Here we will be searching for two delimiters, instead of only one. If both are found (M will no longer be equal to zero when this happens), then the loop is terminated.

Line 145 checks to make sure that a valid number of seconds has been input. If S is less than 0 or greater than 59, then the routine is exited.

```
100  REM   ANALYZE HOURS, MINUTES, AND SECONDS

105  IF LEN (A$) < 6 THEN 195

110  IF RIGHT$ (A$,1) <  > "A" AND RIGHT$ (A$,1) <  > "
     P" THEN 195

120  H = VAL (A$):M = Ø:S = Ø:FOR J = 1 TO LEN (A$):IF
     VAL (MID$ (A$,J,1)) = Ø AND MID$ (A$,J,1) <  > "Ø"
      THEN M = S:S = J + 1:IF M <  > Ø THEN J = LEN (A$
     )

130  NEXT :IF M = Ø THEN 195

140  M = VAL (MID$ (A$,M)):IF M < Ø OR M > 59 THEN 195

145  S = VAL (MID$ (A$,S)):IF S < Ø OR S > 59 THEN 195

150  IF H < 1 OR H > 12 THEN 195

160  A$ = RIGHT$ ("ØØ" + STR$ (H),2) + ":" + RIGHT$ ("Ø
     Ø" + STR$ (M),2) + ":" + RIGHT$ ("ØØ + STR$ (S),2)
      + "" + RIGHT$ (A$,1)

170  T = H * 6Ø + M:IF T > 719 THEN T = T - 72Ø

180  IF RIGHT$ (A$,1) = "P" THEN T = T + 72Ø

185  T = T * 6Ø + S

190  RETURN

195  A$ = "":RETURN
```

Listing 7-2

Table 7-3. Entry and Exit Variables for Time Routine (Hours, Minutes and Seconds)

On Entry:	On Exit:
A$—String to be analyzed Format: HH:MM:SS T HH—Hours MM—Minutes SS—Seconds T—'A' or 'P'	A$—Return string Format: HH:MM:SS T HH—Hours MM—Minutes SS—Seconds T—'A' or 'P' If equal to " " (null) then error in input string. H—Hours M—Minutes S—Seconds T—Total Seconds

Table 7-4. Variable Table for Time Routine (Hours, Minutes and Seconds)

Variable	Type	Purpose	Used in Lines
A	String	Entry/Exit	105, 110, 120, 140, 145, 160, 180, 195
H	Numeric	Hours	120, 150, 160, 170
J	Numeric	Loop Counter	120
M	Numeric	Minutes	120, 130, 140, 160, 170
S	Numeric	Seconds	120, 145, 160, 185
T	Numeric	Total Seconds	170, 180, 185

Line 160 now allows for the inclusion of seconds in the formatted output string. Now, if "1:0:0A" was the input string, then line 160 will format it to be "01:00:00 A".

Finally, line 185 converts all of the calculated minutes to seconds. Notice that in order to get ready for any manipulations, we use the smallest common denominator in the calculated totals. If we are working with only hours and minutes, as in the previous example, then we converted everything to minutes. If seconds are included, then everything must be converted to seconds.

MANIPULATING TIME

Once the "time strings" have been converted to similar units of time, they can then be manipulated just like any other numbers. The

most frequent manipulations, of course, would be subtraction or addition. When subtracting, be careful to use the absolute value of the result. Differentials in time are almost always expressed as positive numbers. It seems difficult to have a −327 minutes. (Unless, of course, you work for NASA.)

After you have manipulated the figures to your satisfaction, then you may want to convert back to hours, minutes and seconds. This is very easy to do, as shown by Listing 7-3 and Tables 7-5 and 7-6.

This routine allows for all three gradations of time. If you were only working to the nearest minutes, then you could substitute M for S, and H for M, and then drop the final original references to H. An alternate solution would be to multiply N by 60 prior to entering this routine. This would convert the total minutes to total seconds.

```
100  S = N:M = INT (S / 60):S = S - (M * 60)

110  H = INT (M / 60):M = M - (H * 60)

120  A$ = STR$ (H) + ":" + RIGHT$ ("00" + STR$ (M),2) +
       ":" + RIGHT$ ("00" + STR$ (S),2)

130  RETURN
```

Listing 7-3

Table 7-5. Entry and Exit Variables for Convert to Time Routine

On Entry:	On Exit:
N—Value to be converted	A$—Formatted time string H—Hours M—Minutes N—Original number S—Seconds

Table 7-6. Variable Table for Convert to Time Routine

Variable	Type	Purpose	Used in Lines
A	String	Formatted time	120
H	Numeric	Hours	110
M	Numeric	Minutes	100, 110
N	Numeric	Input figure	100
S	Numeric	Seconds	100, 110

TIME OUTPUT

Printing times is easy if the above routines are used. In this way, the formatted time is always in A$. The only thing remaining to be done is to determine where to print A$. For printing reports, you may find Chapter 6 useful.

If the hours and minutes routine is used, the output is always seven characters long. Output from the hours, minutes, and seconds routine is always ten characters long.

Fig. 7-3. Mother Time.

Working with times is easy if you do all of your input, analysis, manipulation, and output in modular subroutines. Many programs can be enhanced by the use of time and time-related figures, most notably those that require direct time input from the user, such as payroll, time keeping, or cost-accounting.

In the next chapter we will look at upper and lower cases, and everything in between.

CHAPTER 8

Character Cases

This could be the beginning of a great (but cheap) detective novel. *Character Cases I Have Known and Loved.* Sounds like a best seller, right? Well, while this won't be quite that dramatic, it will hopefully be very useful in your programming efforts.

Many Apples are now equipped to handle both upper and lower case characters. The new Apple IIe's have a full character set included, and quite a number of older Apples have taken advantage of lower case adapters and 80-column boards. Indeed, there are many ways to achieve lower case capability on an Apple, regardless of the model.

THE PROBLEM IN CASE

The problem however comes in string handling, particularly in ordering, because any given string may contain both upper and lower case characters. For example, let's say that you have a program which accepts customer names from a user, and then later you need to search that data for a specific name. The following is a partial list of the name strings that the file may contain:

BAKER, MARTHA
Doe, John
JOHNSON, DOUGLAS
Jones, Barbara
Mitchell, Dennis
Pease, Debra
Porter, Thomas
Thackery, Richard
Thomas, Albert
WYATT, ALLEN

Overall, the list looks about average. There are some entries where the names are all caps, and some mixed. This is not a problem in itself but a problem may surface when you try to search for specific information. For instance, if you ask a user to enter a name to look

for, and he enters "THOMAS, Albert", his search may result in problems. This is because the name, as it appears in the file, is really "Thomas, Albert". This may sound a bit confusing, but hopefully the next few paragraphs will give you a clearer idea of the problem.

At this point it would be helpful to look at an ASCII character chart. Notice in Fig. 8-1 that the ASCII character set includes all kinds of characters. We are just interested, however, in the Alphameric (letters only) characters, both upper and lower case.

Each character has a corresponding number value so that the com-

ASCII CODE TABLE

DEC	HEX	CHAR	KEYBOARD
0	00	NULL	ctrl-@
1	01	SOH	ctrl-A
2	02	STX	ctrl-B
3	03	ETX	ctrl-C
4	04	ET	ctrl-D
5	05	ENQ	ctrl-E
6	06	ACK	ctrl-F
7	07	BEL	ctrl-G
8	08	BS	ctrl-H
9	09	HT	ctrl-I
10	0A	LF	ctrl-J
11	0B	VT	ctrl-K
12	0C	FF	ctrl-L
13	0D	CR	ctrl-M
14	0E	SO	ctrl-N
15	0F	SI	ctrl-O
16	10	DLE	ctrl-P
17	11	DC1	ctrl-Q
18	12	DC2	ctrl-R
19	13	DC3	ctrl-S
20	14	DC4	ctrl-T
21	15	NAK	ctrl-U
22	16	SYN	ctrl-V
23	17	ETB	ctrl-W
24	18	CAN	ctrl-X
25	19	EM	ctrl-Y
26	1A	SUB	ctrl-Z
27	1B	ESCAPE	ESC
28	1C	FS	n/a
29	1D	GS	ctrl-shift-M
30	1E	RS	ctrl-
31	1F	US	n/a
32	20	SPACE	SPACE
33	21	!	!
34	22	"	"
35	23	#	#
36	24	$	$
37	25	%	%
38	26	&	&
39	27	'	'
40	28	(	(
41	29	)	)
42	2A	*	*
43	2B	+	+
44	2C	,	,
45	2D	–	–
46	2E	.	.
47	2F	/	/
48	30	0	0

ASCII CODE TABLE CONT.

DEC	HEX	CHAR	KEYBOARD
49	31	1	1
50	32	2	2
51	33	3	3
52	34	4	4
53	35	5	5
54	36	6	6
55	37	7	7
56	38	8	8
57	39	9	9
58	3A	:	:
59	3B	;	;
60	3C	<	<
61	3D	=	=
62	3E	>	>
63	3F	?	?
64	40	@	@
65	41	A	A
66	42	B	B
67	43	C	C
68	44	D	D
69	45	E	E
70	46	F	F
71	47	G	G
72	48	H	H
73	49	I	I
74	4A	J	J
75	4B	K	K
76	4C	L	L
77	4D	M	M
78	4E	N	N
79	4F	O	O
80	50	P	P
81	51	Q	Q
82	52	R	R
83	53	S	S
84	54	T	T
85	55	U	U
86	56	V	V
87	57	W	W
88	58	X	X
89	59	Y	Y
90	5A	Z	Z
91	5B	[	n/a
92	5C	\	n/a
93	5D	]	shift-M
94	5E	^	^
95	5F	—	n/a

Fig. 8-1. ASCII character chart.

puter can manipulate it. The computer uses these numbers instead of the letters themselves.

Fig. 8-2 shows the numeric conversion of both the user's entry and the file entry for Albert Thomas. The computer uses the numbers, not letters, to make comparisons. By comparing the numbers, it is obvious that the two strings do not match.

It should be obvious by now that the only way to make the computer "think" like a human in this instance might be to convert both strings so that they have the greatest likelihood of being matched. This is accomplished by converting both the source and object strings to the same case, either upper or lower.

```
User:    T   H   O   M   A   S   ,   A   l   b   e   r   t

Codes:   84  72  79  77  65  83  44  65  108 98  101 114 116


File:    T   h   o   m   a   s   ,   A   l   b   e   r   t

Codes:   84  104 111 109 97  115 44  65  108 98  101 114 116
```

Fig. 8-2. Names in ASCII code.

LOWER TO UPPER CASE

By using ASCII numeric representation of data instead of actual letters, it is possible to quickly manipulate the characters to the form we need. The subroutine shown in Listing 8-1 uses this principle to convert mixed case input to all upper case. Variables for the subroutine are defined in Tables 8-1 and 8-2.

This subroutine takes the original string (A$), and converts it one character at a time to upper case. Line 100 sets the resultant string (B$) to null, and also checks to make sure that the input string is at least one character in length.

Line 110 starts the loop that derives the ASCII value of each suc-

```
100  B$ = "":IF LEN (A$) = 0 THEN 150
110  FOR J = 1 TO LEN (A$):A = ASC (MID$ (A$,J,1))
120  IF A < 97 OR A > 122 THEN 140
130  A = A - 32
140  B$ = B$ + CHR$ (A):NEXT J
150  RETURN
```

Listing 8-1

Table 8-1. Entry and Exit Variables for Upper Case Converter Routine

On Entry:	On Exit:
A$—String to convert	A$—Original string B$—Upper case string

Table 8-2. Variable Table for Upper Case Converter Routine

Variable	Type	Purpose	Used in Lines
A	String	Input string	100, 110
A	Numeric	ASCII values	110, 120, 130, 140
B	String	Return string	100, 140
J	Numeric	Loop counter	110, 140

cessive character in A$. Line 120 then checks to see if the character is lower case (remember the ASCII chart). If it isn't lower case, then the actual case conversion is skipped. If it is lower case, then line 130 subtracts 32 from the ASCII value of the character. This effectively changes the character from lower to upper case. Finally, line 140 converts the number back to a character and adds it to B$. The loop is repeated until the process is finished, and the routine returns to the caller in line 150.

While this routine is rather short, it is very effective. It is also useful, because it is much easier to compare strings when everything is either upper or lower case, rather than both.

In our earlier example, we could have used this routine on the search name input by the user to ensure that it was all upper case. Then, as each name was extracted from the name list, we could have used this routine on it to do a quick conversion. Comparisons would then be more likely to succeed. To expand on this point, the conversion routine shown in Listing 8-2 could be used in this way. (Note that variables are summarized in Table 8-3.)

Notice that this program is written with the names in DATA statements. It could just as easily have been written to extract information from a disk file.

UPPER TO LOWER CASE

If we wanted to convert everything to lower case, only two lines would need to be changed in this routine. In Listing 8-3, the same routine is modified to convert to lower case, and the variables are summarized in Tables 8-4 and 8-5.

```
10   DIM A$(10):RESTORE
20   FOR J = 1 TO 10:READ A$(J):NEXT
30    GOTO 1000
100  B$ = "":IF LEN (A$) = 0 THEN 150
110  FOR J = 1 TO LEN (A$):A = ASC (MID$ (A$,J,1))
120  IF A < 97 OR A > 122 THEN 140
130  A = A - 32
140  B$ = B$ + CHR$ (A):NEXT J
150  RETURN
200  A$ = "":B$ = ""
210  GET A$:A$ = LEFT$ (A$,1):A = ASC (A$)
220  IF A = 13 THEN PRINT :RETURN
230  IF A = 8 AND LEN (B$) > 0 THEN B$ = LEFT$ (B$,LEN
     (B$) - 1):PRINT A$;" ";A$;
240  IF A = 8 THEN 210
250  IF A < 32 OR A > 90 THEN 210
260  PRINT A$;
270  B$ = B$ + A$: GOTO 210
1000 HOME
1010 PRINT "ENTER NAME TO SEARCH FOR: ";:GOSUB 200
1020 IF B$ = "" THEN END
1030 E$ = B$:A$ = B$:GOSUB 100:N$ = B$
1040 FOR K = 1 TO 10:A$ = A$(K):GOSUB 100:IF N$ <  > B$
      THEN NEXT : GOTO 1100
1050 PRINT :PRINT "NAME ENTERED: ";E$
1060 PRINT "NAME MATCHED: ";A$(K):PRINT
1070  GOTO 1010
1100 PRINT :PRINT "SORRY, NAME NOT FOUND!":PRINT : GOTO
      1010
1999 END
2000 DATA "BAKER, MARTHA","Doe, John"
2010 DATA "JOHNSON, DOUGLAS","Jones, Barbara"
```

Listing 8-2

```
2020 DATA "Mitchell, Dennis","Pease, Debra"
2030 DATA "Porter, Thomas","Thackery, Richard"
2040 DATA "Thomas, Albert","WYATT, ALLEN"
```

Listing 8-2 cont.

Table 8-3 Variable Table for Case Conversion Sample Program

Variable	Type	Purpose	Used in Lines
A	Numeric	ASCII value of keypress	110, 120, 130, 140, 210, 220, 230, 240, 250
A	String	Entry String	110, 200, 210, 230, 260, 270, 1030, 1040
A(*)	String	Names Array	10, 20, 1040, 1060
B	String	Work String	100, 140, 200, 230, 270, 1020, 1030, 1040
E	String	What User Entered	1030, 1050
J	Numeric	Loop Counter	20, 110, 140
K	Numeric	Loop Counter	1040, 1060
N	String	Converted Entry	1030, 1040

```
100  B$ = "":IF LEN (A$) = 0 THEN 150
110  FOR J = 1 TO LEN (A$):A = ASC (MID$ (A$,J,1))
120  IF A < 65 OR A > 90 THEN 140
130  A = A + 32
140  B$ = B$ + CHR$ (A):NEXT J
150  RETURN
```

Listing 8-3

Table 8-4. Entry and Exit Variables for Lower Case Converter Routine

On Entry:	On Exit:
A$—String to convert	A$—Original string B$-Lower case string

Table 8-5. Variable Table for Lower Case Converter Routine

Variable	Type	Purpose	Used in Lines
A	String	Input string	100, 110
A	Numeric	ASCII values	110, 120, 130, 140
B	String	Return string	100, 140
J	Numeric	Loop counter	110, 140

Notice that the only two lines modified were lines 120 and 130. By changing these, we can check to see if the character being examined is upper case. If it is, then we add 32 to the decimal ASCII value in order to make it lower case.

Converting to lower case will work just as well for comparison purposes. As stated before, the only prerequisite is that both strings being compared are of the same case, either upper or lower.

CASE CONCLUSIONS

This type of routine will work well on an Apple II or II+, but becomes increasingly useful on the Apple IIe or Franklin computers. Since these have upper and lower case built in, searching for information may be more difficult without a routine such as this.

While it is not impossible to correctly search or sort without one of these routines, their use makes a program much more professional and "user-friendly."

CHAPTER 9

Standard File I/O

File handling has baffled many would-be programmers for years. In reality, file handling on the Apple requires nothing more than an extension of keyboard input.

In this chapter we will not be going into highly advanced file handling. File structure and in-depth file development are better left to other technical books, many of which cover data base and data file techniques rather well.

We will, however, attempt to keep things short and simple because, for the majority of file handling needs, short and simple is adequate. If you are doing nothing more than reading and writing random access or sequential files, this chapter should definitely help you organize your file handling tasks.

There are only a few simple rules to keep in mind when planning to use files. These are rules designed to make file handling easier, and they apply to all types of files, whether they be sequential or random access.

1. Always keep a record of how the file was planned and developed.
2. Use array variables wherever possible. The code to handle arrays is generally more compact.
3. Create subroutines for both reading from and writing to a file.
4. If you are using multiple files, open each file only as you need it. When you have either read or written the specific record, close the file immediately. This prevents confusion and stray characters being written to a file.
5. If the file will be more than a few records long, always use random access files. In the long run, they are easier to work with.

Each of these items will be examined in this chapter. Notice that these items fall into two general categories. These are planning and implementation.

PLANNING

The first item of business is to define a file to use as an example. Fig. 9-1 shows a file description covering the file we will be using.

Notice several things. First of all, we have planned for all of the pertinent items surrounding the file. General items such as file name, type, and record length have all been specified. Then the fields of each record are described in detail.

Have you ever written a program that used files, put it on the shelf for several months, and then tried to pick it up and develop it some more? If you have (I know I have), then you can probably appreciate this admonition: Written planning and documentation of files cannot be stressed enough. If they are not done, chances are very good that either you or someone else will be confused when the program is referred to a year from now. If all aspects of the file, plus any related information, are written down as the development happens, then you can save a lot of time in the long run.

Secondly, notice that a variable array was used for the record variables. This is very advantageous because you will be able to use a FOR/NEXT loop to read or write the record. This is faster, more efficient, and less confusing. An extreme example demonstrates the advantage. Suppose that each file record consisted of 50 fields. By naming these individually, it could take quite a few lines of code to both read and write each record. Conversely, by using a 50-element vari-

```
File Name:  TEST FILE

File Type:  Random Access

Record Length:  128

Fields:

Variable        Length      Purpose

 A$(1)            15        First Name

 A$(2)            15        Last Name

 A$(3)            30        Address Line 1

 A$(4)            30        Address Line 2

 A$(5)            33        Comments
```

Fig. 9-1. Sample file specifications.

able array, the chore is made very manageable. In fact, it is likely that the reading and writing can be done in only one or two lines of code.

Third, it should be pointed out that the total of the field lengths does not equal the total record length of 128 bytes. This is because you must add 1 byte per field for the carriage return included at the end of each field. The field lengths total to 123 bytes, but we added 5 (1 for each field) for the carriage returns, and this adds up to 128 bytes.

Finally, notice that the total record length is 128 bytes. This was planned because that number divides evenly into 256. It is best to plan record sizes to fall either to 16, 32, 64, 128, 256, 512, or some other number of bytes that represents a multiple of 256. What, you might well ask, is so "magical" about the number 256? Good question! That is the number of bytes in a disk sector. When even factors or multiples of that number are used, it is easier and faster for DOS to load the sector and transfer the information requested. DOS can deal with other record lengths, but if we help DOS, it will reward us by speeding up the transferral of the information.

IMPLEMENTATION

The first step in implementing file handling actually doesn't have anything to do with the Disk Operating System. It deals with a fact of Applesoft. On page 120 of the Applesoft manual, it points out that the most important speed hint for a program is to use variables instead of constants. The same applies to files. If DOS commands are defined as variables at the beginning of your program, your job of managing the files will be greatly simplified. Also, your program will execute faster. Listing 9-1 and Table 9-1 show how this could be achieved.

```
10   D$ = CHR$ (4):OP$ = D$ + "OPEN":CL$ = D$ + "CLOSE"
     :WR$ = D$ + "WRITE":RD$ = D$ + "READ"

20   FO$ = "TEST FILE,D1,L128":FU$ = "TEST FILE,R"
```

Listing 9-1

Table 9-1. Variable Table for File Command Instructions

Variable	Type	Purpose
CL	String	DOS CLOSE command
FO	String	File name for opening file
FU	String	File name file operations
OP	String	DOS OPEN command
RD	String	DOS READ command
WR	String	DOS WRITE command

Notice that all of the DOS commands that will be used have been defined as variables. If it was determined that we would need other DOS commands, such as APPEND or POSITION, then those would also be defined. The two versions of the file name with appended information (such as drive and length) have also been included as variables.

Now it is time to set up the routines that will actually handle reading and writing specific file records. The routine in Listing 9-2 will handle reading a file record. As you study this routine, and its variables, which are summarized in Tables 9-2 and 9-3, notice how short it is.

```
100  REM  - READ FILE RECORD
110  PRINT OP$;FO$:PRINT RD$;FU$;RN
120  FOR J = 1 TO 5:INPUT A$(J):NEXT
130  PRINT CL$:RETURN
```

Listing 9-2

Table 9-2. Entry and Exit Variables for File Read Routine

On Entry:	On Exit:
RN—Record number to read NOTE: This routine assumes that all DOS values have been set as described earlier in this chapter.	A$(*)—Record variable values

Table 9-3. Variable Table for File Read Routine

Variable	Type	Purpose	Used in Lines
A(*)	String	Record variable	120
CL	String	CLOSE command	130
FO	String	File name	110
FU	String	File name	110
J	Numeric	Loop counter	120
OP	String	OPEN command	110
RD	String	READ command	110
RN	Numeric	Record number	110

This routine does quite a bit. All of it is done efficiently and easily, however, because we took the time to define variables early in the program. Then all we did was put the variables together for the commands we wanted.

Writing to a file record is much the same process. The routine in Listing 9-3 (with variables summarized in Tables 9-4 and 9-5) will write to a file record. Notice that there are only changes in two lines from the read routine.

In both reading and writing file records, the process has been made very simple. With only minor modifications, this same process can be applied to any file. Notice, also, that we have used only string variables. It is a good rule of thumb never to input numeric variables if it can be avoided. For a discussion of the reasons for this, refer to Chapter 2.

```
150  REM  - WRITE FILE RECORD
160  PRINT OP$;FO$:PRINT WR$;FU$;RN
170  FOR J = 1 TO 5:PRINT A$(J):NEXT
180  PRINT CL$:RETURN
```

Listing 9-3

Table 9-4. Entry and Exit Variables for File Write Routine

On Entry:	On Exit:
RN—Record number to write NOTE: This routine assumes that all DOS values have been set as described earlier in this chapter.	A$(*)—Record variable values

SEQUENTIAL FILES

So far the examples have only used random access files, but the same principles can be applied to sequential files. The only differences would be to take away references to lengths and record numbers. If these changes were implemented in the routines outlined so far, they would appear as in Listing 9-4, with variables defined as in Table 9-6.

Notice that the routines to handle sequential file I/O are not that much different from those that handle random access I/O. A change was made in line 20 for the file names. With sequential files it is not necessary to include the length parameters. In addition, lines 110 and

Table 9-5. Variable Table for File Write Routine

Variable	Type	Purpose	Used in Lines
A(*)	String	Record variable	170
CL	String	CLOSE command	180
FO	String	File name	160
FU	String	File name	160
J	Numeric	Loop counter	170
OP	String	OPEN command	160
RN	Numeric	Record number	160
WR	String	WRITE command	160

160 were changed so as not to specify a record number, since you do not need to specify a record when utilizing sequential files. Apart from these changes, however, the subroutines for sequential and random access files are the same.

```
10   D$ = CHR$ (4):OP$ = D$ + "OPEN":CL$ = D$ + "CLOSE"
     :WR$ = D$ + "WRITE":RD$ = D$ + "READ"

20   FO$ = "TEST FILE,D1":FU$ = "TEST FILE"

30    GOTO 1000

100  REM  - READ FILE RECORD

110  PRINT OP$;FO$:PRINT RD$;FU$

120  FOR J = 1 TO 5:INPUT A$(J):NEXT

130  PRINT CL$:RETURN

150  REM  - WRITE FILE RECORD

160  PRINT OP$;FO$:PRINT WR$;FU$

170  FOR J = 1 TO 5:PRINT A$(J):NEXT

180  PRINT CL$:RETURN

1000 REM  - CONTROLLING CODE
```

Listing 9-4

Table 9-6: Variable Table for Sequential File Routines

Variable	Type	Purpose	Used in Lines
A(*)	String	Record variable	120, 170
CL	String	CLOSE command	10, 130, 180
FO	String	File name	20, 110, 160
FU	String	File name	20, 110, 160
J	Numeric	Loop counter	120, 170
OP	String	OPEN command	10, 110, 160
RD	String	READ command	10, 110
WR	String	WRITE command	10, 160

CHAPTER 10

Sorting

Sorting is a field unto itself. Entire volumes of books have been written to discuss various sorting algorithms. There are specialists that do nothing except work with sorting programs. This is not one of those books, nor am I one of those people. I doubt if you are one of those people either. Neither of us has to be unless we are planning a bright future in Systems Design and Sort Technique.

If you are like me, all you need to do is periodically sort a short- to medium-length list of information. BASIC implementations of popular sorting algorithms will do fine for these applications. It should be mentioned up front, however, that under various conditions, BASIC sorting tends to be rather slow. Machine language sorting is much faster by comparison. If speed is not one of your major concerns, however, then BASIC routines will work just fine.

There are two general approaches to sorting. These are either the "in-memory" (internal) sorts and the "disk" (external) sorts. The former type is well suited for most BASIC applications of short- to medium-length lists. Longer lists will require a sectional sorting technique with intermediate data "saves" to disk. This latter type of sorting, because of its advanced nature, will not be discussed in this chapter, or even in this book. If a discussion of more advanced sorting applications is desired, then it is suggested that you visit your local computer store or library. Sorting techniques have not changed considerably over the last several years, so you should be able to find an adequate book on the subject in either of these places.

SORTING FUNDAMENTALS

The main idea behind sorting is to place items in a predetermined order based on a common quality. For instance, if a gym teacher had

a class of 30 students, he might want to have them line up in alphabetical order according to their last name. Or he may want to line them up according to height, or age. Name, height, and age are called *keys,* as they determine in what order the resultant line-up, or list, will be.

Once a key is determined, then you will also need to know if the list is desired in ascending or descending order. In other words, will the list be ordered from least to greatest value, or from greatest to least? In the case of the gym teacher, ascending order would be from A to Z, shortest to tallest, or youngest to oldest, depending on the key that was selected. If he decided to arrange the line in descending order, then it would be just the reverse of the ascending ordered line.

The sorting process itself is nothing more than comparing items of the original unordered list and exchanging them with other items in order to get the list closer to its final sorted position. The number of elements to be sorted is usually expressed by the variable N.

There have been many sorting algorithms designed, each supposedly more efficient than preceding methods. Each sorting method is adept at general list sorting, and each sorting algorithm is outstanding in sorting an unordered list that meets certain requirements. Some algorithms, for example, might be great at sorting partially ordered lists, or changing ascending ordered lists to descending ordered lists. However, these same algorithms may do poorly when used to sort randomly distributed lists. The actual sorting method that you choose should depend, in large part, on the type and order of data that you need to sort.

There are minor differences between sorting lists of numbers and lists of alphanumeric data. For the sake of expediency, however, all examples in this chapter will deal only with sorting strings. If you wish to apply the same techniques to numeric data, all you need to do is use numeric variables in the place of the string variables.

When you are deciding on a sort routine to use, you will have to be the final judge as to what fits your needs best. Certain "specialty" sorts may work better for your needs. In this chapter, however, we will only be discussing a few of the many algorithms available. Each of these will be general-purpose sorting algorithms. The first to be discussed will be the Substitution Sort.

SUBSTITUTION SORTING

The simplest sorting technique is the Substitution Sort, sometimes called a "Bubble Sort". This technique works by comparing each string to every other string in the list to find which one belongs where. As each comparison is made, it is determined whether a "switch" of the data at the two locations should occur or not. If so,

then the values are exchanged and the process continues. Each time a complete pass is made through the unsorted list, one more element is positioned in its correct sorted position. Therefore, we would assume that one pass of the list is required for each element in the original list. Thus, if the original list contains eight elements, then eight passes will be required to correctly sort the list.

Actually, that many passes are not required in application. For a list of eight elements, only seven passes are required, becuase the last pass actually places the final two elements in their correct positions. An eighth pass would be redundant. Fig. 10-1 demonstrates the way a Substitution Sort would work with a list of eight elements.

Notice that each time through the list, an element at the front of the list is placed in correct position. We can then ignore these elements, because we know that they are positioned correctly and need no more comparisons. This means that the first time through the list there are seven comparisons required, six the second time, five the third,

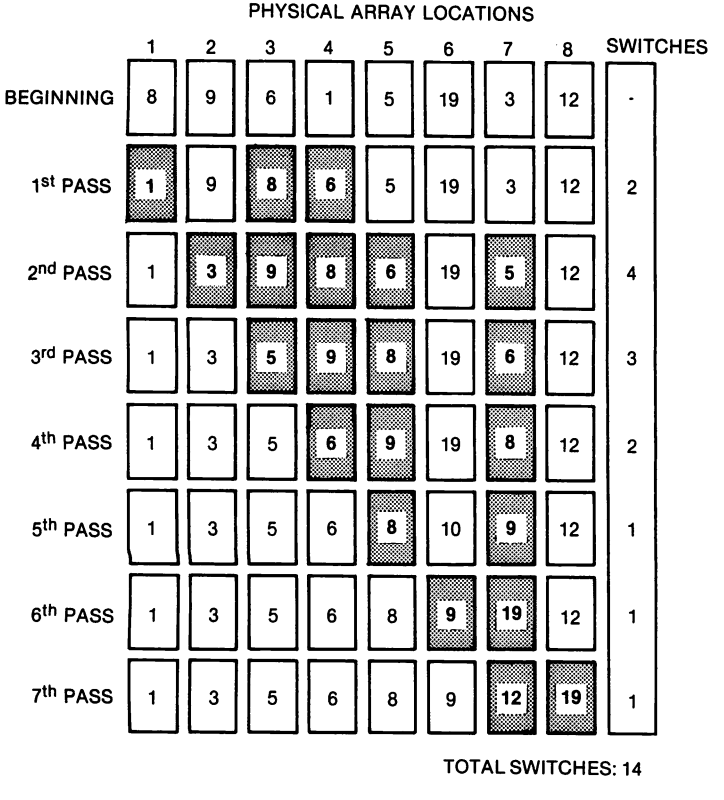


Fig. 10-1. Sorting sequence for substitution sort.

and so on until all passes have been completed. The number of comparisons for a Substitution Sort, then, is **(N-1)!** (pronounced N-1 Factorial). This is a mathematical way of saying what was expressed earlier. This concept of the number of comparisons required is displayed in Fig. 10-2.

Implementing the Substitution Sort is simple by use of FOR/NEXT loops. Listing 10-1 shows a routine which will sort an alphanumeric array in ascending order. The variables for the routine are listed in Tables 10-1 and 10-2.

This routine is not long at all. It is the "quick and dirty" type of sort that will work best on short lists. Notice that there is no wasted time in the routine, because there are no instances where a string is compared to itself. The two loops are always offset by one array element to allow for the most efficient use of this algorithm.

As we pointed out earlier, there is no need to compare the elements of the array that we know have been correctly placed. Thus, the K loop (which does the actual comparisons) uses J+1 as a bottom boundary on each iteration.

The drawback of this routine is that all this "switching" causes it to run rather slowly. If you had an array of 100 elements, each time through the array it is only possible to have one element be put in order. However, it is highly possible that there were 5, 10, 20, or more switches in order to place that one string. This tends to slow things down a bit because when that much string manipulation is done, Applesoft has to do quite a bit more overhead work, such as "garbage collection." That is the process of freeing up string storage space periodically.

```
(N-1)! = 7 + 6 + 5 + 4 + 3 + 2 + 1 = 28
```

Fig. 10-2. The formula for the number of comparisons required by the substitution sort.

```
100  FOR J = 1 TO N - 1
110  FOR K = J + 1 TO N
120  IF S$(J) > S$(K) THEN T$ = S$(J):S$(J) = S$(K):S$(
     K) = T$
130  NEXT :NEXT :RETURN
```

Listing 10-1

Table 10-1. Entry and Exit Variables for Substitution Sort Routine

On Entry:	On Exit:
N—Highest element of array S$(*)—Array to be sorted	S$(*)—Sorted array

Table 10-2. Variable Table for Substitution Sort Routine

Variable	Type	Purpose	Used in Lines
J	Numeric	Loop counter	100, 110, 120
K	Numeric	Loop counter	110, 120
N	Numeric	Highest element of array	100
S(*)	String	Array to sort	120
T	String	Temporary string	120

SPEEDING UP SUBSTITUTION

As the list of elements to be sorted becomes longer, it proves increasingly critical to speed up the sorting process. This can be done in any of three ways. You can decrease the number of comparisons, decrease the number of exchanges, or use different variables. The best way to speed up the Substitution Sort is to use different variables (which in effect reduces the number of exchanges necessary).

Applesoft tends to work quite a bit faster with numeric variables, since there is no garbage collection required, and there are quite a few less actual bytes to move around. If the sorting routine were modified to allow for the switching of numeric pointers instead of the actual strings, then it would theoretically be faster. It would be possible to improve the efficiency to only one string switch per pass, maximum. This is done by using pointers, and by only switching strings at the end of the K loop.

This (Listing 10-2) is how the Substitution Sort would appear if we were to introduce these modifications. Tables 10-3 and 10-4 list variables.

The biggest change here is the addition of J1 as a string pointer. It starts in line 100 as being equal to J, and then is reset equal to K each time the routine encounters a string of lesser value than the one to which it previously pointed. This is continued until the "K loop" is finished, and then the actual string switching takes place in line 130. In order to save more time, the switching only takes place if a switch

```
100  FOR J = 1 TO N - 1:J1 = J
110  FOR K = J + 1 TO N
120  IF S$(J1) > S$(K) THEN J1 = K
130  NEXT :IF J1 <  > J THEN T$ = S$(J):S$(J) = S$(J1):
     S$(J1) = T$
140  NEXT :RETURN
```

Listing 10-2

Table 10-3. Entry and Exit Variables for Modified Substitution Sort Routine

On Entry:	On Exit:
H—Highest element of array S$(*)—Array to be sorted	S$(*)—Sorted array

Table 10-4. Variable Table for Modified Substitution Sort Routine

Variable	Type	Purpose	Used in Lines
J	Numeric	Loop counter	100, 110, 130
J1	Numeric	Low string pointer	100, 120, 130
K	Numeric	Loop counter	110, 120
N	Numeric	Highest element of array	100
S(*)	String	Array to sort	120, 130
T	String	Temporary string	130

is needed. In other words, there is only a switch if J1 does not equal J. If this check was not done, it would be possible to switch in place which would waste time and string space.

Functionally, this routine is the same as the straight Substitution Sort. Successive array elements are compared to find the lowest valued elements. Then they are placed in their correct sorted order. The difference is that they are not placed in that order *until* each pass is completed. Thus, where it was possible (with a list of 100 items) to have 10, 20, or 30 exchanges per pass before, we will only have 1 exchange per pass with this modified version of the algorithm.

Earlier it was noted that this routine is quite a bit faster than the unmodified Substitution Sort. Listing 10-3 should illustrate this. The first part of the program creates an array of 100 random string ele-

```
100  DIM S$(100),T$(100):N = 100
110  FOR J = 1 TO N
120  J1 = INT (RND (2) * 15):IF J1 < 2 THEN 120
130  FOR K = 1 TO J1
140  K1 = RND (2) * 91:IF K1 < 32 OR K1 > 90 THEN 140
150  S$(J) = S$(J) + CHR$ (K1)
160  NEXT :NEXT
200  FOR J = 1 TO N:T$(J) = S$(J):NEXT
280  X = FRE (0)
290  PRINT CHR$ (7);"STARTING SUBSTITUTION SORT"
300  FOR J = 1 TO N - 1
310  FOR K = J + 1 TO N
320  IF S$(J) > S$(K) THEN T$ = S$(J):S$(J) = S$(K):S$(
     K) = T$
330  NEXT :NEXT
340  PRINT CHR$ (7);"DONE WITH SUBSTITUTION SORT"
380  X = FRE (0)
390  PRINT CHR$ (7);"STARTING MODIFIED SUBSTITUTION SOR
     T"
400  FOR J = 1 TO N - 1:J1 = J
410  FOR K = J + 1 TO N
420  IF T$(J1) > T$(K) THEN J1 = K
430  NEXT :IF J1 <  > J THEN T$ = T$(J):T$(J) = T$(J1):
     T$(J1) = T$
440  NEXT
450  PRINT CHR$ (7);"DONE WITH MODIFIED SUBSTITION SORT
     "
```

Listing 10-3

ments. The second part of the program then makes a second array equal to the first. Finally, each of the duplicate arrays is sorted. One is sorted by the regular substitution sort, and the other is sorted by the modified version. As each section of the program is completed, a notice is displayed, and the bell is sounded so that you can use your watch to time each sorting method. (The variables for this listing are summarized in Table 10-5.)

Table 10-5. Variable Table for Substitution Sort Test Program

Variable	Type	Purpose	Used in Lines
J	Numeric	Loop counter	110, 150, 200, 300, 310, 320, 400, 410, 430
J1	Numeric	Low element pointer	400, 410, 430
K	Numeric	Loop counter	130, 310, 320, 410, 420
K1	Numeric	ASCII character	140, 150
N	Numeric	Upper array limit	100, 110, 200, 300, 310, 400, 410
S(*)	String	Array to sort	100, 150, 200, 320
T	String	Temporary string	320, 430
T(*)	String	Array to sort	100, 200, 420, 430
X	Numeric	Free space	280, 380

In case you didn't have a watch handy, I'll tell you what mine told me. By timing both methods, using duplicate arrays to sort, and working under the same conditions, you'll find the modified routine runs in about half the time needed for the regular substitution sort.

Obviously, if these routines are implemented in another program, the speed of the routines may differ dramatically. The speed is dependent on many things, including the size of the program, the number of string variables in memory, the number of items to sort, the ordering of the original list, and whether the routine is "straight-line" or a subroutine. In this example, we made the sorting algorithms straight-line in order to attempt making them as quick as possible.

SHELL SORT

This algorithm is different from the Substitution Sort introduced earlier. It was introduced in 1959 by D. L. Shell (thus the name) in July of 1959. It differs from the Substitution Sort in that it relies on comparisons and exchanges of array elements that are not immediate neighbors. Comparisons are done at a predetermined distance between elements. For example, if the distance between elements was 4, then element 1 is compared with element 5, 2 with 6, 3 with 7, and so on.

Much has been said and written about Shell Sort, so I will not go into detail in this chapter. The greatest debate has been over the determination of an appropriate starting distance between items to be compared. Shell, in his original algorithm, used an interval equal to one-half of the total number of elements in the array. Each successive pass cut the distance in half, until an interval of only one element was achieved. This required only a small number of passes to sort a relatively large array. As an example, it only takes one more pass to sort 200 items than it does to sort 100 items.

In Listing 10-4 we shall use the distance proposed by Shell. Tables 10-6 and 10-7 list variables for this implementation of Shell Sort.

The distance between comparisons for each pass of the array is set in line 510. Initially, this distance is equal to one-half of the total number of elements in the array. In lines 520 through 550, we execute a loop that runs the actual comparisons and exchanges string values if necessary.

Shell Sort is relatively simple, yet it offers significant speed advantages over the earlier sorts that were introduced. It is best used on short- to medium-length lists of data.

SHELL SORT COMPARISON

The next step, of course, is to compare the performance of Shell Sort to the Substitution Sorts under similar conditions. The program in

```
500  I = N

510  I = INT (I / 2):IF I = 0 THEN 570

520  FOR J = I TO N:T$ = U$(J)

530  FOR K = J - I TO 1 STEP  - I:IF T$ <  = U$(K) THEN
      U$(K + I) = U$(K):NEXT K

540  U$(K + I) = T$

550  NEXT J

560   GOTO 510

570  RETURN
```

Listing 10-4

Table 10-6. Entry and Exit Variables for Shell Sort Algorithm

On Entry:	On Exit:
N—Highest element of array U$(*)—Array to be sorted	U$(*)—Sorted array

Table 10-7. Variable Table for Shell Sort Algorithm

Variable	Type	Purpose	Used in Lines
I	Numeric	Distance pointer	500, 510, 520, 530, 540
J	Numeric	Loop counter	520, 530, 550
K	Numeric	Loop counter	530, 540
N	Numeric	Highest array element	500, 520
T	String	Temporary	520, 530, 540
U(*)	String	Sort array	520, 530, 540

Listing 10-5, which uses the variables in Table 10-8, has had the Shell Sort algorithm appended, and gives the same type of output as the earlier comparison program. Get your watches ready.

Notice that the actual execution of the Substitution Sorts has slowed down somewhat in this comparison. The coding was not changed, but the memory of the computer has more variables to contend with. This added overhead, as noted earlier, can slow down performance of sorting routines.

Table 10-9 shows the comparative times for a sample run of this program. Because the random strings may be generated differently on each run, your times may be different than those noted.

So far, we have looked at how to speed up a sort by reducing the number of actual switches. The next step would be to speed it up by reducing the total number of comparisons required. This would take a completely different type of sorting algorithm, however. The one that shall be introduced here is called Quicksort.

QUICKSORT

This sorting algorithm is quite elegant. It was devised by C.A.R. Hoare in 1962. The idea behind Quicksort is to exchange non-adjacent elements of an array in the hopes of achieving a more nearly sorted array. Partitioning is used to accomplish this sorting. This may sound confusing, but it actually works as implemented.

For example, suppose that you have the same gym class we introduced earlier in the chapter. Class members were lined up in no specific order, and the gym teacher wanted to have them lined up in order according to height. If he were trying to do this in the same method that Quicksort does it, then he would divide (partition) the class in half, and compare members from each half of the class. If the class consisted of 30 members, then a member from the bottom half

```
100  N = 100:DIM S$(N),T$(N),U$(N)

110  FOR J = 1 TO N

120  J1 = INT (RND (2) * 15):IF J1 < 2 THEN 120

130  FOR K = 1 TO J1

140  K1 = RND (2) * 91:IF K1 < 32 OR K1 > 90 THEN 140

150  S$(J) = S$(J) + CHR$ (K1)

160  NEXT :NEXT

200  FOR J = 1 TO N:T$(J) = S$(J):U$(J) = S$(J):NEXT

280  X = FRE (0)

290  PRINT CHR$ (7);"STARTING SUBSTITUTION SORT"

300  FOR J = 1 TO N - 1

310  FOR K = J + 1 TO N

320  IF S$(J) > S$(K) THEN T$ = S$(J):S$(J) = S$(K):S$(
     K) = T$

330  NEXT :NEXT

340  PRINT CHR$ (7);"DONE WITH SUBSTITUTION SORT"

380  X = FRE (0)

390  PRINT CHR$ (7);"STARTING MODIFIED SUBSTITUTION SOR
     T"

400  FOR J = 1 TO N - 1:J1 = J

410  FOR K = J + 1 TO N

420  IF T$(J1) > T$(K) THEN J1 = K

430  NEXT :IF J1 <  > J THEN T$ = T$(J):T$(J) = T$(J1):
     T$(J1) = T$

440  NEXT

450  PRINT CHR$ (7);"DONE WITH MODIFIED SUBSTITION SORT
     "

480  X = FRE (0)

490  PRINT CHR$ (7)"STARTING SHELL SORT"

500  I = N

510  I = INT (I / 2):IF I = 0 THEN 570

520  FOR J = I TO N:T$ = U$(J)

530  FOR K = J - I TO 1 STEP  - I:IF T$ <  = U$(K) THEN
      U$(K + I) = U$(K):NEXT K
```

Listing 10-5

```
540  U$(K + I) = T$
550  NEXT J
560   GOTO 510
570  PRINT CHR$ (7)"DONE WITH SHELL SORT"
```

Listing 10-5 cont.

Table 10-8. Variable Table for Sort Comparison Program

Variable	Type	Purpose	Used in Lines
I	Numeric	Distance pointer	500, 510, 520, 530, 540
J	Numeric	Loop counter	110, 150, 200, 300, 310, 320, 400, 410, 430, 520, 530, 550
J1	Numeric	Low element pointer	120, 130, 400, 420, 530, 540
K	Numeric	Loop counter	130, 310, 320, 410, 420, 530, 540
K1	Numeric	ASCII character	140, 150
N	Numeric	Highest array element	100, 110, 200, 300, 310, 400, 410, 500, 520
S(*)	String	Sort array	100, 150, 200, 320
T	String	Temporary	320, 430, 520, 530, 540
T(*)	String	Sort array	100, 200, 420, 430
U(*)	String	Sort array	100, 200, 520, 530, 540
X	Numeric	Free space	280, 380, 480

Table 10-9. Sorting times for comparison.

Sort Type	Sample Time	Improvement
Substitution Sort	79 seconds	—— %
Modified Substitution Sort	39 seconds	50.63 %
Shell Sort	15 seconds	58.97 %

is compared to a member of the top half. The gym teacher knows that the bottom half of the class will contain all of the shorter members. If comparison of the two members shows the taller member in the bottom half of the class, then a trade is made in order to at least get the members in the right half of the class. This process continues until the entire class is in correct order.

This example, of course, has been greatly generalized and oversimplified. When using arrays, Quicksort is more complex. It continues to partition the list down until there is only one element per partitioned sub-list. Then it works its way back up the partitions until they are all done. If each partition is in order, then the entire array will be in order.

To accomplish the sort, we must choose some element of the array to be placed in its correct final position. Then the remaining elements are arranged so that they are either less than or greater than the original chosen element.

This chosen element is called a *pivot,* and there are many theories on the best way to choose it for optimal performance of Quicksort. The best performance will occur when the value of the pivot is the median value of the partition being sorted. In practice, however, there is no way to ensure this without extensive testing. Such testing can slow down the overall performance of the sort, particularly with a medium-length list. For our purposes, we shall always use the first sequential element of the partition as the pivot.

Once the pivot is selected, the next step is to scan forward from there until we find a value greater than the pivot value. Then we scan backward from the end of the partition until a value is found that is less than the value of the pivot. Then we exchange the lower and higher values, since they are both in the theoretical "wrong half" of the partition.

This process is continued until the forwards and backwards comparison pointers pass each other. At this point, the pivot value and the final forward value are exchanged, and it is assumed that the pivot value is now in its final sorted position. Then the whole process of selecting a pivot is repeated again and again until the whole array is completely sorted.

For those of you who are mathematically inclined, perhaps an explanation using variables would be helpful. Assume that the lower and upper bounds of our alphanumeric array are P and Q, such that it appears as U$(P) . . . U$(Q). On the first pass through the array with Quicksort, the pivot point string, denoted as X$, will be equal to U$(P). We will use I and J as our scanning variables. These are respectively set as equal to P+1 and Q. Then we start a process of incrementing I to look for an array element whose value is greater than or equal to X$. When we find one, then we start decrementing J

until we find an array element whose value is less than or equal to X$. When that is found, we switch U$(I) and U$(J). If J is still greater than I, then we go back to the point where I was being incremented to look for the next array value higher than or equal to X$. The process is repeated until J is less than I, at which point X$ and U$(I) are switched in position.

This may all sound quite a bit more confusing than it actually is. Fig. 10-3 shows this process graphically with an array of only 8 elements. If you keep studying the process, it will become more and more clear as you go.

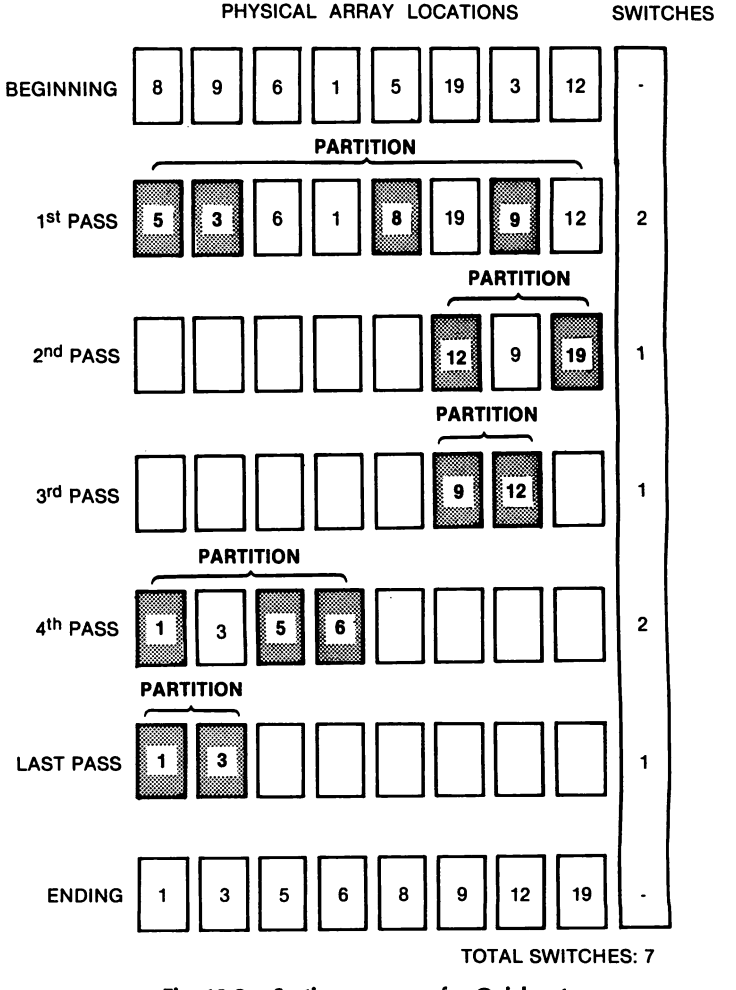


Fig. 10-3. Sorting sequence for Quicksort.

Listing 10-6 shows the BASIC implementation of a Quicksort variation, and Tables 10-10 and 10-11 show the variables used.

While this routine takes more code than the Substitution and Shell Sorts, it still is rather compact. By studying this listing, it should be fairly clear as to how Quicksort is implemented. Take particular care to apply this listing to the process shown in Fig. 10-3.

Line 610 does nothing more than set up the entry variables for the routine. P is set equal to the lowest element of the array, and Q is set equal to the highest. T0 is a stack pointer, and is initially set to zero. The stack is used to save the beginning and ending pointers to partitions that still need to be sorted.

Line 620 checks to see if the current partition is done being sorted. If it is (P > = Q) then execution skips to line 710 where the routine checks to see if another unfinished partition is waiting in the wings.

Lines 630 through 670 are the heart of the actual sort. These lines carry out the process of selecting a pivot (V$) and then scanning forward from the front of the partition for elements that are less than or equal to the pivot, and backwards from the back of the partition for elements that are greater than or equal to the pivot.

When both of these are found, then elements are switched in line 660, and this process is continued until the pointers (I and J) "pass"

```
600  REM  - QUICKSORT

610  P = 1:Q = N:T0 = 0

620  IF P >  = Q THEN 710

630  V$ = V$(P):I = P:J = Q + 1

640  J = J - 1:IF V$(J) > V$ THEN 640

650  I = I + 1:IF V$(I) < V$ AND I < N THEN 650

660  IF J > I THEN T$ = V$(I):V$(I) = V$(J):V$(J) = T$:
      GOTO 640

670  V$(P) = V$(J):V$(J) = V$

680  IF (J - P) < (Q - J) THEN ST(T0 + 1) = J + 1:ST(T0
      + 2) = Q:Q = J - 1: GOTO 700

690  ST(T0 + 1) = P:ST(T0 + 2) = J - 1:P = J + 1

700  T0 = T0 + 2: GOTO 620

710  IF T0 <  > 0 THEN Q = ST(T0):P = ST(T0 - 1):T0 = T
     0 - 2: GOTO 620

720  RETURN
```

Listing 10-6

Table 10-10. Entry and Exit Variables for Quicksort Algorithm

On Entry:	On Exit:
N—Highest element of array ST(*)—Stack array (empty, but dimensioned to proper size) V$(*)—Array to be sorted	V$(*)—Sorted array

Table 10-11. Variable Table for Quicksort Algorithm

Variable	Type	Purpose	Used in Lines
I	Numeric	Forward counter	630, 650, 660
J	Numeric	Backward counter & loop counter	630, 640, 660, 670, 680, 690
N	Numeric	Total array elements	610, 650
P	Numeric	Lower partition pointer	610, 620, 630, 670, 680, 690, 710
Q	Numeric	Upper partition pointer	610, 620, 630, 680, 710
ST(*)	Numeric	Stack	680, 690, 710
T	String	Temporary	660
T0	Numeric	Stack pointer	610, 680, 690, 700, 710
V	String	Temporary	630, 640, 650, 670
V(*)	String	Sort array	630, 640, 650, 660, 670

each other. When they do, then the values of the pivot and U$(J) are exchanged.

Next, a new partition is determined, and the process repeated. The partition not currently being sorted is saved on the stack for future reference.

QUICKSORT COMPARISON

This routine works very quickly for most sorting needs. It is interesting, however, to compare it to the earlier sorting methods. By using the same comparison program that was introduced earlier, and appending Quicksort, we can get an idea of how the routine fares. List-

```
100  N = 100:DIM S$(N),T$(N),U$(N),V$(N)

110  FOR J = 1 TO N

120  J1 = INT (RND (2) * 15):IF J1 < 2 THEN 120

130  FOR K = 1 TO J1

140  K1 = RND (2) * 91:IF K1 < 32 OR K1 > 90 THEN 140

150  S$(J) = S$(J) + CHR$ (K1)

160  NEXT :NEXT

200  FOR J = 1 TO N:T$(J) = S$(J):U$(J) = S$(J):V$(J) =
      S$(J):NEXT

280  X = FRE (0)

290  PRINT CHR$ (7);"STARTING SUBSTITUTION SORT"

300  FOR J = 1 TO N - 1

310  FOR K = J + 1 TO N

320  IF S$(J) > S$(K) THEN T$ = S$(J):S$(J) = S$(K):S$(
     K) = T$

330  NEXT :NEXT

340  PRINT CHR$ (7);"DONE WITH SUBSTITUTION SORT"

380  X = FRE (0)

390  PRINT CHR$ (7);"STARTING MODIFIED SUBSTITUTION SOR
     T"

400  FOR J = 1 TO N - 1:J1 = J

410  FOR K = J + 1 TO N

420  IF T$(J1) > T$(K) THEN J1 = K

430  NEXT :IF J1 <  > J THEN T$ = T$(J):T$(J) = T$(J1):
     T$(J1) = T$

440  NEXT

450  PRINT CHR$ (7);"DONE WITH MODIFIED SUBSTITION SORT
     "

480  X = FRE (0)

490  PRINT CHR$ (7)"STARTING SHELL SORT"

500  I = N

510  I = INT (I / 2):IF I = 0 THEN 570

520  FOR J = I TO N:T$ = U$(J)

530  FOR K = J - I TO 1 STEP  - I:IF T$ <  = U$(K) THEN
      U$(K + I) = U$(K):NEXT K
```

Listing 10-7

```
540  U$(K + I) = T$
550  NEXT J
560   GOTO 510
570  PRINT CHR$ (7)"DONE WITH SHELL SORT"
580  DIM ST(SQR (N) + 1)
590  X = FRE (0)
600  PRINT CHR$ (7)"STARTING QUICKSORT"
610  P = 1:Q = N:T0 = 0
620  IF P >  = Q THEN 710
630  V$ = V$(P):I = P:J = Q + 1
640  J = J - 1:IF V$(J) > V$ THEN 640
650  I = I + 1:IF V$(I) < V$ AND I < N THEN 650
660  IF J > I THEN T$ = V$(I):V$(I) = V$(J):V$(J) = T$:
      GOTO 640
670  V$(P) = V$(J):V$(J) = V$
680  IF (J - P) < (Q - J) THEN ST(T0 + 1) = J + 1:ST(T0
      + 2) = Q:Q = J - 1: GOTO 700
690  ST(T0 + 1) = P:ST(T0 + 2) = J - 1:P = J + 1
700  T0 = T0 + 2: GOTO 620
710  IF T0 <  > 0 THEN Q = ST(T0):P = ST(T0 - 1):T0 = T
     0 - 2: GOTO 620
720  PRINT CHR$ (7)"FINISHED WITH QUICKSORT"
```

Listing 10-7 cont.

ing 10-7 will illustrate the comparison. (Note the program variables which are listed in Table 10-12.)

The approximate results of this program are shown in Table 10-13. These times were derived with the use of my handy-dandy wrist watch, so those of you with deluxe chronographs may get more accurate times. The point is, however, that we have developed several sorting techniques that work quite well.

Again, your actual results in programs may vary depending on other program factors. When you run the program, you may notice a decrease in time efficiency for all of the earlier algorithms. Such is the price of added overhead! Also, notice that the times shown for Shell Sort and Quicksort in Table 10-13 are equal. This does not mean that they do equally well, only that they did equally well on this test. The

Table 10-12. Variable Table for Sort Comparison Program

Variable	Type	Purpose	Used in Lines
I	Numeric	Distance pointer & forward counter	500, 510, 520, 530, 540, 630, 650, 660
J	Numeric	Loop counter & backward counter	110, 150, 200, 300, 310, 320, 400, 410, 430, 520, 530, 550, 630, 640, 660, 670, 680, 690
J1	Numeric	Low element pointer	120, 130, 400, 420, 430
K	Numeric	Loop counter	130, 310, 320, 410, 420, 530, 540
K1	Numeric	ASCII character	140, 150
N	Numeric	Total array elements	100, 110, 200, 300, 310, 400, 410, 500, 520, 580, 610, 650
P	Numeric	Lower partition pointer	610, 620, 630, 670, 680, 690, 710
Q	Numeric	Upper partition pointer	610, 620, 630, 680, 710
S(*)	String	Sort array	100, 150, 200, 320
ST(*)	Numeric	Stack	580, 680, 690, 710
T	String	Temporary	320, 430, 520, 530, 540, 660
T(*)	String	Sort array	100, 200, 420, 430
T0	Numeric	Stack pointer	610, 680, 690, 700, 710
U(*)	String	Sort array	100, 200, 520, 530, 540
V	String	Temporary	630, 640, 650, 670
V(*)	String	Sort array	100, 200, 630, 640 650, 660, 670
X	Numeric	Free space	280, 380, 480, 590

Table 10-13. Sorting times for comparison.

Sort Type	Sample Time	Improvement
Substitution Sort	79 seconds	—— %
Modified Substitution Sort	38 seconds	51.90 %
Shell Sort	15 seconds	60.53 %
Quicksort	15 seconds	0.00 %

Quicksort algorithm is more efficient, and will show a performance increase as the size of the array to be sorted increases.

SORTING CONCLUSIONS

The Quicksort algorithm works very well on medium to long lists. For shorter lists, the modified Substitution Sort or the Shell Sort may work better. You may want to experiment to find out which type of algorithm will work best for you.

If you enjoy working with sorting techniques, and sorting intrigues you, then you may want to search out some additional material on the subject. Your local library is a good place to start. Computer magazines frequently carry articles on sorting.

CHAPTER 11

Program Menus

If you walk into a restaurant (and the hostess doesn't ignore you), you will be promptly seated and handed a menu. Without a menu, it's extremely difficult to find out what the restaurant has to offer. Restaurants generally spend quite a bit of time, effort, and money on their menus to make a good impression on the customers.

It is just the same with program menus. If you were to write a program without a menu it would be difficult at best for a user to discover how to use the program. The absence of a menu can generate a negative feeling about your program. If a restaurant spends so much time working on a menu, shouldn't you?

MENU COMPONENTS

Menus have several parts. Generally, there is a series of choices displayed on the screen. Then, there is a way for the user to enter the option he wishes to initiate.

In this chapter, we will be going over each section of a menu, then presenting a routine that will handle all aspects of menu display and selection. This will be a full-screen menu. Many programs on the market today have partial-screen menus, but these are not generally as clear as those with full-screen menus.

Fig 11-1 shows a sample menu from a current commercial program. In function, this is somewhat similar to the type of menu we will develop at the end of this chapter, although our actual design will be different.

MENU CHOICES

The choices that make up your menu should be clear, concise, and to the point. They should represent all of the main functions and op-

Fig. 11-1. Sample menu.

erations of your program. For instance, you may have developed a program that files names of individuals. In a menu for such a program, you would decide on what choices to offer. They may be simple, basic operations, such as:

1. Enter Names
2. Change Names
3. Delete Names
4. Exit Program

Notice that all of the basic functions are covered here. Also notice that this menu offers a way to exit the program. Programs within menus that do not offer a way to exit the program tend to be rather confusing, not to mention frustrating, for the user.

This menu could be expanded if we wanted other operations to be included. For example, if we decided to add a print and display routine, we could expand the menu to signify the additional choices.

It is not necessary that every function in a program be outlined on one menu. It is quite logical, and indeed quite acceptable, to have sub-menus from a main menu. This allows for your program to be broken up into distinct, logical units which give an overall impression of program organization.

For example, we may have several operations that perform routine chores, such as purging files, backing up disks, creating new disks, displaying system parameters, etc. Instead of putting each of these on the main menu, why not allocate a menu just for these items, and access it from a main menu choice called "System Operations"? This is logical, and it shows the user that these are important system functions, although they are not in the "mainstream" of program activity.

MENU DISPLAY

This area of menu development is just as important, and in many instances more difficult than deciding on the choices to be offered on the menu.

Most menus are displayed on one screen. It is cumbersome to have a menu occupy more than one screen. The first item on a menu is the title, or heading. It lets the user know what the choices represent. This heading might be nothing more than the title of the program, or it may be the heading for the section of the program currently executing.

The heading should contain this title, as a minimum. It serves as a road sign for the novice user, and it also reassures the experienced operator. Other items, such as underlines, or the date, can also be included in a heading, depending on the needs of the computer users.

The next portion of the display contains the actual choices. The choices should be displayed in a clear, readable fashion; however, their clarity and readability may vary from program to program depending on their length and complexity. For instance, there may be five choices for the menu, but each choice could be rather long. There is a difference between how "EXIT" would be displayed, and how "TRANSVERSE POLYNOMIAL COMPUTATION" would be displayed. If you center one horizontally, then the other might look awkward. You must attempt to strike a happy medium.

The actual choices should be set off from the left margin of the screen by at least a few spaces. This sets them off from the heading, and gives the screen a more "balanced" appearance. The routine presented later in this chapter starts all choices at the seventh print position on each line, which makes the menu look rather nice on the screen.

Try to keep the screen neat and uncluttered. Listing 40 possible commands on the screen may confuse users, particularly the first-time user. Well-planned displays are inviting and don't scare people off.

This demonstrates why striking a happy medium makes good sense. One of the causes of computer-phobia is a screen that jumps out at the user instead of inviting him in. Too many choices on the

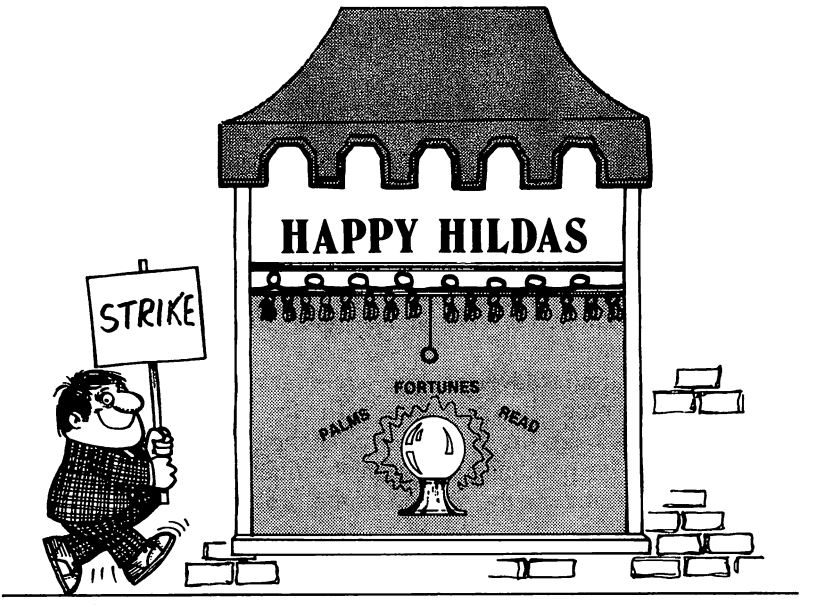


Fig. 11-2. Striking a happy medium.

screen can be intimidating. Conversely, if not all of the available operations are shown on the menu, the user may feel lost. It looks like a job for Hilda, the Happy Medium! Use her influence in defining the appearance of your menu screens.

CHOICE SELECTION

Selecting an option from a menu should be as easy as pressing one or two keys. Single-key input is easy enough for the programmer to provide, and it adds a nice touch.

Single-key choice selection is not difficult if you remember to limit the number of choices on a menu. For instance, it is a good idea to limit the number of choices to 9. If you go over 9, *group several* choices into sub-menus. Then all a user needs to do is press a number between 1 and 9 to initiate a selection.

The prompt for a choice should be short and to the point. *Function Choice:* or *Enter Selection:* may sound rather un-imaginative, but they do communicate what is desired of the user. These will be excellent prompts, as long as action other than pressing a key is required of the user. If your routine requires that the user press RETURN to ini-

tiate the option, then you may need to include that information as part of your prompt or general display. 'Tis best not to leave the folks hangin'. Translated, that means "Think how you would feel if you were in the user's place." If there is a possibility of confusion or bewilderment, it is best to clarify the choices available. The routine at the end of this chapter requires that the user press RETURN. All necessary information is displayed on the menu.

The actual placement of the prompt line depends on how the rest of the menu is structured. The standard place is near the bottom of the screen. This lets a user's eyes travel from the choices to the prompt in a linear fashion, instead of bouncing around the screen to find out what's next.

THE MENU ROUTINE

The routine in Listing 11-1 is structured to present a series of up to nine choices for a user, and then to elicit a response. It allows the user to either enter a number from the screen or to use the left- and right-arrow keys to position the screen cursor. You may have seen

```
100  TEXT :HOME :HTAB 11:PRINT "MAIN PROGRAM MENU":HTAB
      5:FOR J = 1 TO 10:PRINT "---";:NEXT :PRINT :VTAB
     22:PRINT "  <-- --> TO SELECT, RETURN TO ACCEPT":P
     OKE  - 16368,0:CH = 0

110  FOR K = 1 TO NP:VTAB 6 + K:POKE 36,6:PRINT K")";:P
     OKE 36,9:PRINT M$(K):NEXT

120  VTAB 6 + CH:POKE 36,6:INVERSE :PRINT CH;:POKE 36,9
     :PRINT M$(CH):NORMAL :TD = CH

130  KP = PEEK ( - 16384):IF KP < 128 THEN 130

140  KP = KP - 128:POKE  - 16368,0

150  IF KP = 8 THEN CH = CH - 1:IF CH < 1 THEN CH = NP

160  IF KP = 21 THEN CH = CH + 1:IF CH > NP THEN CH = 1


170  IF KP = 13 THEN RETURN

180  IF KP = 8 OR KP = 21 THEN 210

190  KP = KP - 48:IF KP < 1 OR KP > NP THEN PRINT CHR$
     (7): GOTO 130

200  CH = KP

210  IF CH = TD THEN 130

220  VTAB 6 + TD:POKE 36,6:PRINT TD;:POKE 36,9:PRINT M$
     (TD): GOTO 120
```

Listing 11-1

similar menus on commercial products such as Magic Window and MultiPlan. Although MultiPlan's menu is similar, it is only a partial screen menu. Variables for this routine are summarized in Tables 11-1 and 11-2.

Table 11-1. Entry and exit variables for menu generator subroutine

On Entry:	On Exit:
M$(*)—Arrray containing menu choices NP—Number of menu choices	CH—Menu choice desired.

Table 11-2. Variable table for menu generator subroutine

Variable	Type	Purpose	Used in Lines
CH	Numeric	Menu choice desired	100, 120, 150, 160, 200, 210
J	Numeric	Loop counter	100
K	Numeric	Loop counter	110
KP	Numeric	Keypress	130, 140, 150, 160, 170, 180, 190, 200
M(*)	String	Menu text	110, 120, 220
NP	Numeric	Number of menu choices	110, 150, 160, 190
TD	Numeric	Last choice	120, 210, 220

Notice that there are several items that are required before entering this routine. You must set up the actual wording of the menu choices in an array (M$(*)). For example, if you wanted "ENTER NAMES" to be your first menu choice, then you would assign M$(1) equal to "ENTER NAMES". This would be done for each of your menu choices. Also, you need to set the upper limit pointer, NP equal to the number of choices on the menu. This is used for error and boundary checking.

When the routine returns to the caller, the value of the function chosen will be assigned to CH. This will be a number between 1 and whatever upper limit you have assigned to NP. At this point, you can use a statement such as **ON CH GOTO X, Y, Z** to branch to the various function handlers. Of course, X, Y, and Z should all be replaced with appropriate line numbers.

In out next chapter, we will deal with error routines in a program.

CHAPTER 12

Error Handling

All things in life have a potential for mistakes. That is why there are erasers on pencils and error codes on computers. Knowing how to handle an eraser is one thing; knowing how to handle an error on the computer is another. In this chapter we will discuss specific errors and error handling techniques.

Error messages are the words that appear on your screen when an error occurs in the program you are executing, or when you type in something which is unacceptable to the computer. Error codes, on the other hand, are the numbers that the computer uses to represent each of those error conditions that may occur during operation. Error codes come into direct play when using ONERR GOTO routines, as will be discussed later in the chapter.

Error messages and codes are divided into two distinct groups. There are error codes for DOS, and there are those for Applesoft. DOS generates error messages that have codes in the range of 1 through 15. Applesoft error messages straddle this range. There is an error code 0, and then there are error codes between 16 and 255. The DOS error codes are consecutively numbered, while Applesoft error codes are not.

APPLESOFT ERRORS

When an Applesoft error occurs in a program, a message is displayed on the screen and program execution stops. This can be rather disconcerting to the user. Careful debugging of a program can weed out most of the possible Applesoft error possibilities. If the error occurs in a program, then program execution stops, and the error message is displayed, along with the line number of where the error occurred.

Fig. 12-1 shows the possible Applesoft error messages. All of these messages and codes can be found in the Applesoft manual (pages 115-117 and 136), but not much information is given on what causes them. Furthermore, they are not very organized in the manual. Both the manual's list (page 136) and the descriptions (pages 115-117) are incomplete. We will go over each error here in hopefully a little more detail.

NEXT WITHOUT FOR is generated when a NEXT statement is encountered for which no corresponding FOR statement was found. This will also happen when the variable used with the NEXT statement (such as NEXT J) was not the variable used with the FOR statement.

Code	Message
Ø	NEXT WITHOUT FOR
16	SYNTAX
22	RETURN WITHOUT GOSUB
42	OUT OF DATA
53	ILLEGAL QUANTITY
69	OVERFLOW
77	OUT OF MEMORY
9Ø	UNDEFINED STATEMENT
1Ø7	BAD SUBSCRIPT
12Ø	REDIMENSIONED ARRAY
133	DIVISION BY ZERO
149	ILLEGAL DIRECT
163	TYPE MISMATCH
176	STRING TOO LONG
191	FORMULA TOO COMPLEX
21Ø	CAN'T CONTINUE
224	UNDEFINED FUNCTION
254	Bad Response To An INPUT
255	CTRL-C Interrupt Attempted

Fig. 12-1. Applesoft errors.

SYNTAX means that you have entered something that the computer doesn't understand, and you should go back and do it again.

RETURN WITHOUT GOSUB occurs when a RETURN statement is encountered before a corresponding GOSUB has occurred.

OUT OF DATA means that too many READ statements have been executed in the program. Each READ causes the DATA text pointer to be incremented to the next item of DATA. If there is no more DATA, then this message or code is generated.

ILLEGAL QUANTITY means that you have asked Applesoft to work with a number (quantity) that is not legal for the type of operation you want. This can occur when you use negative numbers in the wrong context as subscripts for an array, for a square root seed value, or in a string manipulation. It can also occur if improper values are being used in some math functions such as LOG or exponents.

OVERFLOW means that you have tried to use a number that is too big for Applesoft to work with.

OUT OF MEMORY is a grievous error which means either that you are using too many variables (particularly strings), your program is too large to allow for variable space, or you have simply overextended the bounds of your machine's memory.

UNDEF'D STATEMENT is shorthand for UNDEFINED STATEMENT. It means that you are referencing a line number that either does not exist or that Applesoft cannot find. (Technically there may be no distinction between these conditions, but in practice there may be.)

BAD SUBSCRIPT is generated when you try to reference an array element which is outside the set boundaries of the array. It usually occurs when you have not dimensioned an array large enough.

REDIM'D ARRAY is, again, shorthand for REDIMENSIONED ARRAY. Once you have dimensioned an array, either explicitly or by the implied method, you cannot later redimension it. Implied dimensioning means referring to an array element less than 10 at some point in your program. Without a specific dimensioning statement, this will automatically dimension the array to 10 elements.

DIVISION BY ZERO means just that. You tried to divide a number by zero. As we all know (from math classes), that is impossible in the real world. Computers generally like to operate in the real world.

ILLEGAL DIRECT can only happen in direct mode, not while a program is running. It is generated when you try to use statements like INPUT, GET, DATA, or DEF FN from immediate mode. These will only work under program operation.

TYPE MISMATCH is generated when you try to assign a numeric value to an alphanumeric variable, or vice-versa. It also happens if you enter the wrong type of variable (numeric or alphanumeric) as a parameter for another function, and it is not what the function ex-

pected or needed. For example, PRINT LEFT$(A$,B$) will not work, and will generate a Type Mismatch error.

STRING TOO LONG means that you tried to assign an alphanumeric variable to a group of characters over 255 characters in length. This usually occurs when you are concatenating strings.

FORMULA TOO COMPLEX is generated when a formula that you have entered is too big or complex for Applesoft to correctly interpret and execute.

CAN'T CONTINUE only happens in immediate mode when you try to use CONT to resume a program, and Applesoft can't comply.

UNDEF'D FUNCTION means that you tried to use a function (FN) that you didn't define first. You can prevent this by making sure that all functions are defined at the front of a program.

Bad Response to an INPUT Statement is not really an error message, it is an explanation of an error code. Error code 254 has no corresponding error message, as it is generated only while a program is running with an active ONERR GOTO statement. It means that you (or your disk file) responded to an INPUT's request for a numeric variable with an alphanumeric answer. This won't work. If you tried this with no ONERR GOTO statement active, then all you would get is a ?REENTER message. This is not really an error message, but more of a request for reentry by Applesoft.

Ctrl C Interrupt Attempted is again an explanation, not an actual error message. Error code 255 is generated only when a program is running with an active ONERR GOTO statement. If no such statement is in effect, then a Control-C is handled by termination of program execution and the message BREAK IN XX, where XX is the line number where the interrupt occurred.

APPLESOFT ERROR MESSAGE TABLE

All of the messages contained in Fig. 12-1 are contained in Applesoft so that they can be printed when an error is encountered. Listing 12-1 (for which variables are summarized in Table 12-1) will print out the error code and the corresponding error message from this text table within Applesoft itself.

Notice that the only thing consistent about the error codes is that they increase in value. As you have probably discerned from looking at the program, the error codes that Applesoft generates are actually "offsets" into the error message text table. For example, the message NEXT WITHOUT FOR begins the text table (at relative location 0), so it is error code 0. The next message (SYNTAX) occurs starting with the 16th character of the table, so it is error code 16.

All that this program does is calculate these offsets, print them, and then retrieve and print (character by character) the error message that

```
5   HOME
10  PRINT "CODE","ERROR MESSAGE"
20  PRINT "----","--------------------"
30  X = 53855
40  Z = X - 53855:IF Z > 225 THEN 90
50  PRINT RIGHT$ ("   " + STR$ (Z),4),
60  X = X + 1:Y = PEEK (X):PRINT CHR$ (Y);
70  IF Y > 127 THEN PRINT : GOTO 40
80   GOTO 60
90  PRINT :END
```

Listing 12-1

Table 12-1. Variable Table for Applesoft Error Text Table Printer

Variable	Type	Purpose	Used in Lines
X	Numeric	Memory Pointer	30, 40, 60
Y	Numeric	ASCII Character	60, 70
Z	Numeric	Error Pointer	40, 50

is in the text table at that point. Notice that the text table begins at memory location 53856 ($D260). The end of each individual message is determined by the high bit of the character being set. Thus, if the value of the character being printed is greater than 127, it is the last character of that error message.

There are two other sets of memory locations that are important to understanding errors and how to deal with them. We will look at those right after we take a quick look at DOS errors.

DOS ERROR CODES

DOS error messages are separate from Applesoft errors because DOS is not an integral part of Applesoft BASIC. It is separate, and as such has its own codes. There are only 15 possible DOS error codes, and 14 possible error messages. There is a discrepancy of one because one of the messages has two possible codes. DOS error codes are in the range of 1 through 15.

There is a detailed description of each of the 14 possible error messages on pages 114 through 122 of the DOS manual. (These page numbers are correct for both the green and red manuals, although

they may be different in the "new" DOS manual that will accompany the Apple IIe.) Because of the excellent explanation of error messages and codes given in the DOS manuals, we will not at this time discuss the reasons they are generated. Fig. 12-2, however, shows all of the possible DOS error messages.

These are the error codes that will be most likely to pop up in a program. Applesoft errors can be weeded out by careful debugging, but there is always a chance for DOS errors if your program provides an interface between DOS commands and the user.

Just as in Applesoft, the actual text of the DOS error messages are saved in a text table within DOS. This table begins at memory location 43377 ($A971). The program in Listing 12-2 shows a way to print each of the codes and its associated message from this text table; Table 12-2 shows the variables used.

Notice that there is an error code 0 that lists out when this program is run. This is not a bonafide error code, but simply an attention-getter. It is a character sequence consisting of a carriage return, bell, carriage return.

Code	Message
1	LANGUAGE NOT AVAILABLE
2	RANGE ERROR
3	RANGE ERROR
4	WRITE PROTECTED
5	END OF DATA
6	FILE NOT FOUND
7	VOLUME MISMATCH
8	I/O ERROR
9	DISK FULL
10	FILE LOCKED
11	SYNTAX ERROR
12	NO BUFFERS AVAILABLE
13	FILE TYPE MISMATCH
14	PROGRAM TOO LARGE
15	NOT DIRECT COMMAND

Fig. 12-2. DOS errors.

```
5   HOME
10  PRINT "CODE","ERROR MESSAGE"
20  PRINT "----","----------------------"
30  X = 43376
40  FOR Z = 0 TO 15
50  PRINT RIGHT$ ("   " + STR$ (Z),3),
60  X = X + 1:Y = PEEK (X):PRINT CHR$ (Y);
70  IF Y < 128 THEN 60
80  IF Z = 2 THEN Z = 3
90  PRINT :NEXT Z
```

Listing 12-2

Table 12-2. Variable Table for DOS Error Text Table Printer

Variable	Type	Purpose	Used in Lines
X	Numeric	Memory Pointer	30, 60
Y	Numeric	ASCII Character	60, 70
Z	Numeric	Error Pointer	40, 50, 80, 90

You will also notice that in the program we have skipped error code 3. This is because DOS, in its normal course of operations, automatically prints the second error message every time error code 3 is called for.

This brings up another interesting point. The error codes for DOS are not handled in the same fashion as those for Applesoft. Those, you will remember, represented offsets in the text table. In DOS, however, an error code is generated by a portion of the program, and then something called a "look-up table" is used to print the proper message from the above text table. The "look-up table" contains nothing but the address of each DOS error message. This address is then used to determine what to print.

ERROR HANDLING

Normally, when an error is encountered, an error message is displayed on the screen and program execution stops. This can be circumvented by use of the Applesoft ONERR GOTO command. This command causes program execution to branch to a certain line

number if any Applesoft or DOS errors occur. The line number can be any valid line number. There should, however, be some sort of routine included to take appropriate care of the error condition.

As stated earlier, there are several sets of machine locations important to error processing. These are represented in Fig. 12-3. They are locations that contain the error code of the error that occurred, as well as the line number where that error occurred. There is also a location to be used to turn error processing (ONERR GOTO) off.

Several of these locations are used at least once in any good error handling routine. This will be seen in the Listing 12-3 and Table 12-3.

There are several things that should be pointed out about this routine. Notice first of all that there is a big jump between the first few lines and the actual error handling routine. This is because those first few lines handle some important ground work for any error handling routine. First, line 10 pokes a machine-language routine into memory that allows certain bugs in Applesoft error handling to be corrected after an error is detected. This procedure is outlined on pages 82 and 136 in the Applesoft manual.

Next, line 20 invokes the actual ONERR GOTO statement that is necessary in order to put the error handling routine into effect. After this line is executed, program control will be shifted to line 50000 upon detection of an error.

You may be wondering why this routine begins at line 50000. This is to ensure that the routine is separate from the rest of the program. You could renumber this routine and put it most anywhere.

MEMORY LOCATION	EXPLANATION & COMMENTS
216	Flag to determine if there is an active ONERR GOTO statement. If this location is equal to zero, then none is active. If greater than 127, then one is in effect.
218/219	These contain the number of an error that has occurred if there is an active ONERR GOTO statement. Line number is in a low / high arrangement.
222	Error code of last encountered error.

Fig. 12-3. Error processing memory locations.

```
10      RESTORE :FOR J = 768 TO 777:READ K:POKE J,K:NEXT

20      ONERR  GOTO 50000

1000    REM  THIS GAP REPRESENTS THE MAIN BODY OF THE PR
        OGRAM

50000   CALL 768:ET = PEEK (222):EL = PEEK (218) + PEEK
        (219) * 256

50010   PRINT CHR$ (4)"PR#0":PRINT CHR$ (4)"CLOSE"

50020   IF ET = 0 OR ET = 16 OR ET = 22 OR ET = 42 OR ET
         = 90 OR ET = 107 OR ET = 120 OR ET = 163 OR ET
        = 191 OR ET = 224 THEN PRINT :PRINT "PROGRAM ERR
        OR #"ET" IN LINE "EL".": GOTO 51000

50030   IF ET = 53 THEN E$ = "ILLEGAL QUANTITY": GOTO 50
        220

50040   IF ET = 69 THEN E$ = "OVERFLOW": GOTO 50220

50050   IF ET = 77 THEN E$ = "OUT OF MEMORY": GOTO 50220

50060   IF ET = 133 THEN E$ = "DIVISION BY ZERO": GOTO 5
        0220

50070   IF ET = 176 THEN E$ = "STRING TOO LONG": GOTO 50
        220

50080   IF ET = 254 THEN E$ = "INPUT": GOTO 50220

50090   IF ET = 255 THEN E$ = "CTRL-C": GOTO 50220

50100   IF ET = 2 OR ET = 3 THEN E$ = "RANGE": GOTO 5022
        0

50110   IF ET = 4 THEN E$ = "WRITE PROTECTED": GOTO 5022
        0

50120   IF ET = 5 THEN E$ = "END OF DATA": GOTO 50220

50130   IF ET = 6 THEN E$ = "FILE NOT FOUND": GOTO 50220

50140   IF ET = 7 THEN E$ = "VOLUME MISMATCH": GOTO 5022
        0

50150   IF ET = 8 THEN E$ = "I/O": GOTO 50220

50160   IF ET = 9 THEN E$ = "DISK FULL": GOTO 50220

50170   IF ET = 10 THEN E$ = "FILE LOCKED": GOTO 50220

50180   IF ET = 11 THEN E$ = "DOS SYNTAX": GOTO 50220

50190   IF ET = 12 THEN E$ = "NO BUFFERS AVAILABLE": GOT
        O 50230

50200   IF ET = 13 THEN E$ = "FILE TYPE MISMATCH": GOTO
        50220
```

Listing 12-3

```
50210  E$ = "UNKNOWN ERROR #" + STR$ (ET): GOTO 50230
50220  E$ = E$ + " ERROR"
50230  PRINT :PRINT E$" IN LINE "EL"."
50240  PRINT :PRINT "PRESS <RETURN> TO CONTINUE...";
50250  GET CH$:IF CH$ <  > CHR$ (13) THEN 50250
50260  PRINT : GOTO 1000
50270  DATA 104,168,104,166,223,154,72,152,72,96
```

Listing 12-3 cont.

Table 12-3. Variable Table for Error Handling Routine

Variable	Type	Purpose	Used in Lines
CH	String	User Keypress	50250
E	String	Error Message	50030, 50040, 50050, 50060, 50070, 50080, 50090, 50100, 50110, 50120, 50130, 50140, 50150, 50160, 50170, 50180, 50190, 50200, 50210, 50220, 50230
EL	Numeric	Error Line	50000, 50020, 50230
ET	Numeric	Error Type	50000, 50020, 50030, 50040, 50050, 50060, 50070, 50080, 50090, 50100, 50110, 50120, 50130, 50140, 50150, 50160, 50170, 50180, 50190, 50200, 50210
J	Numeric	Loop Counter	10
K	Numeric	Data Reader	10

Line 50000 calls the machine language correction routine described earlier. Then the variable ET (Error Type) is set equal to the value of memory location 222. This is the error code memory location, if you remember correctly. Finally, variable EL (Error Line) is set equal to the values in memory locations 218 and 219. This is the line number in which the error occurred.

Next, each error is checked and handled. Notice that there are several errors that are handled as a group. For instance, if an error is caused by wrong or flawed coding on the part of the programmer,

then the message "PROGRAM ERROR" followed by the actual code and line number is all that is displayed.

Notice that at the end of our error handling routine, all execution is transferred to line 1000. This line number is just used as an example. You should set it to the number of either your initializing routine for variables or your main menu area. On certain error codes, you may wish program execution to stop. This can be done also, but if the program will be used by a nonprogrammer, it is a good idea to make sure that you put an instructional message on the screen to tell him what to do next. Usually, when I want the program to stop in case of a dire error, I add a message instructing the user to call his local programmer, including the phone number (if possible). That way I, or the programmer, will know immediately that a serious error has occurred.

As a final note, these error messages may be changed so that the error handling routine prints any messages desired. You can also drop testing for error codes that are unlikely to occur. You should, however, keep a "catch-all" handler in effect at all times. In other words, if an error code is found that does not get trapped out, then the code and line number should be printed before returning to the main program section.

APPENDIX

Tools of the Trade

Many companies try to sell you products that are "guaranteed" to help you program better. Actually, they will only help you if you use them. And when do you use them? When they are really needed, of course!

You don't need every new utility that comes down the pike. There are, however, a few basic ones that will help you to be more productive in your programming efforts. The follwing list may be of help:

1. A program line editor. GPLE is a very good one. It is available through Call −A.P.P.L.E.
2. A complete renumber utility. Apple Computer puts out one that is very adequate.
3. A good disk zap/editor program. Apple-Aids published by Howard Sams is a good one.
4. A good editor/assembler if you do machine coding. There are many on the market.

Obviously this list may vary as your needs vary, but these items are a good place to start. Whenever I am reading magazines, I always keep my eye peeled for any new ones. If they sound like something I would only use once every other year, I pass them by.

There is one other resource that it is almost impossible to do without if you do any programming at all. That is a good reference library. There are only five books that I use in depth. These are:

1. Applesoft II Manual, Apple Computer Inc.
2. The DOS Manual, Apple Computer Inc.
3. Apple II Reference Manual, Apple Computer Inc.
4. Beneath Apple DOS, Worth & Lechner
5. All About Applesoft, Call −A.P.P.L.E.

I find myself referring to them at least once a day. The information contained in them is worth many times the cover price to someone who wants to

make the most of his Apple computer. You will notice that Apple puts out three of the books listed. That is one of the primary reasons that the Apple is the most popular micro on the market. They seem to know the importance of user-friendly documentation.

GLOSSARY

This glossary is designed to be friendly, informative, and in places, humorous. If you are looking for highly technical definitions of terms, you should probably invest in a computer dictionary. This glossary should give you at least enough information to enlighten you on some words and terms that may have seemed foreign to you in the text.

ADDRESS is a unique number associated with a specific computer memory location.

ALGORITHM is a set of instructions to do a defined task. A mathematical example would be to define the task, such as finding the area of a square. The algorithm to do this would be Height multiplied by Width.

ALPHAMERIC is a seldom-used term that denotes a string composed of letter characters only.

ALPHANUMERIC is the term used to describe a series of characters that may be either alphabetic (letters), numeric (numbers), or symbolic (such as control characters).

APPLE is the computer that heralded the dawn of the popular personal computing age. Invented by Steve Wozniak and Steve Jobs in 1976, it is the largest selling personal computer in history.

APPLESOFT is the version of Microsoft's Floating-Point BASIC implemented for use on the Apple II series of personal computers. It was an early effort by Microsoft, with later changes made by Apple to accommodate graphics.

ARGUMENT is what users of competing personal computers usually end up in. It is also the data entered into a program that is used in a calculation or procedure to formulate an output.

ARRAY is an organized arrangement of data items. For instance, the letters FI may represent a single variable, but FI(9) represents the ninth element of a data array. Each element is addressable by changing the subscript within the parenthesis.

ASCEND means to rise, or go up. In sorting, ascending order indicates that the sorted items will be ordered from lowest value to highest value.

ASCII is the American Standard Code for Information Interchange. It is a

method used by the vast majority of mini- and microcomputers to encode characters by arranging the individual bits of a byte.

ASSEMBLER is a program used to translate a series of commands and directives into the actual machine language codes needed to run directly on a computer processor.

AUTO-START ROM is the ROM (see ROM) on an Apple computer that allows an initial program to be loaded and executed when the power is turned on. In most cases, this initial program is DOS (see DOS), which in turn controls the loading and running of an object program.

BACKUP is a term that refers to the process of making a copy of valuable information in case the original copy is damaged or destroyed.

BASE usually refers to a mathematic numbering system. For instance, base 10 (or decimal), allows for 10 different digits (0 through 9) in each number position. In contrast, base 16 (hexadecimal) allows 16 digits (0 through F) per number position.

BASIC stands for Beginner's All-purpose Symbolic Instruction Code. The most widely used computer language in the mini- and microcomputer markets.

BINARY is a number system based on powers of 2. The only digits in binary are "0" and "1". It is of practical use in computers where electronic circuits can only be off ("0") or on ("1").

BIT is a single binary digit. It is also the smallest indivisible unit of information that is understood by a computer.

BOOTING is the term used to describe the process that a computer goes through to start up. This includes loading the Disk Operating System (see DOS), and the program to be first run. With an Apple computer system, booting is accomplished by turning the computer power on, or typing **PR#s**, where 's' is the slot number that contains the disk controller for the disk drive from which you wish to boot.

BRANCH means to change the program execution from the normal contiguous series of steps to another predetermined step. Branching is usually achieved by use of a GOTO or GOSUB instruction from BASIC.

BUG is an undesirable pest that sometimes sneaks into the best of programs. Upon detection, bugs can sometimes hide again and should be documented thoroughly by the finder.

BYTE is a group of bits that collectively represent either a letter (on 8 bit computers) or a word (on larger processors).

CARRIAGE RETURN is the term given to the ASCII code that causes the computer to return to the leftmost column. It is also used to signify an end-of-input. Usually this is the ASCII code 13 ($D). It is generated from a keyboard by pressing the key marked RETURN (ENTER on some computers).

CASE is a term used to describe whether capital letters are used or not. Upper case means a capital letter, the opposite of lower case letters.

CATALOG is the term used to describe the collective list of files that reside on a disk, tape, or some other mass storage device. Sometimes referred to as a directory, it contains the information necessary for the DOS (see DOS) to track and manage each file.

Also a command in Apple DOS that causes a list of disk files to be displayed on the current output device.

CHARACTER is either a letter, space, number or special symbol. Each character requires one byte of computer memory.

CHIP is short for "microchip" and refers to a collection of solid state circuits in one device. Each chip is generally designed and created to perform a specific task.

COMPILE means taking source code and converting it to object code. Translating from a higher-level language (such as Assembler) to instructions that the computer can understand directly (such as machine code).

COMPILER is the program that compiles, or translates, source code to object code. (See COMPILE).

COMPUTER is an electronic device that performs calculations according to predetermined instructions at a very fast rate. Depending on the way the computer is programmed, this may or may not be of use to humans. Computers generally fall into one of three classifications. These are Micros, Minis, and Mainframes.

CONCATENATION is the process of joining two or more alphanumeric strings together to make one string.

CONTROL CHARACTER is a special two-key combination of keystrokes that directs the computer to do something special. On the Apple, control characters are generated by holding down the key marked CTRL and pressing any other alphameric key at the same time. Some control characters are used so frequently that they have been assigned to one specific key. Examples of this are RETURN, LEFT-ARROW, RIGHT-ARROW, and the BREAK key on some computers.

CPU stands for Central Processing Unit, the heart of any computer system.

CROSS-ASSEMBLER is a program that executes under the control of one type of microprocessor to create program code that will ultimately be executed under control of a different microprocessor.

CRT stands for Cathode Ray Tube, and is "computerese" for a video terminal or computer monitor.

CURSOR is a video marker appearing on the computer monitor that lets the user know where the next input or output is to occur. The cursor is generally a small blinking block or underline character.

DATA is nothing more than information. The word "data" is used to save space and ink.

DATA FORMAT is the guidelines and rules that dictate the order, style, and condition of information that must be adhered to when supplying data for a computer program.

DEFAULT is a term used to describe the "assumed" value that is accepted when none is offered in its place.

DEMODULATE is the process of converting signal tones to electrical impulses. Used primarily over phone lines. (See MODEM.)

DESCEND means to fall, or go down. In sorting, it means ordering the elements to be sorted from greatest to least value.

DISKETTE is a semi-permanent storage device for electronic information. Disks come in many different sizes, but most all are round, hence the name "disks." Early attempts by the government to create a square-disk standard failed. Subsequent efforts in this area led to the removal of the center hole and marketing them as Mag-cards for word processors.

The most popular sizes for disk today are 8-inch, 5 1/4-inch, and 3 1/2-inch.

DOCUMENTATION is the set of written or printed instructions that accompany a program to explain how to use it. There has been no successful approach to teach how to use documentation.

DOS is short for Disk Operating System. This is the computer program that regulates all interaction with the disk drives and the information stored there.

DP is the computerese term for Data Processing which is the computerese term for Information Juggling.

ELEMENT, when used in relation to programming, is a single item of a larger data array. For instance F1(9) is a single element variable of a larger array of data.

ENTRY refers to the answers that you give to questions within a program. Most entries are always terminated (ended) with a carriage return. This is accomplished by pressing the RETURN key.
Single key entries, however, should not require a carriage RETURN to terminate.

FIELD is a term used to describe a single piece of information. Names and dates are examples of possible fields.

FILE is a collection of related records. These comprise a logical block of information that has a specific name that may be accessed by a user. File names are contained within the disk catalog (see CATALOG).

FLAG is a marker that is set within a program to mark the presence or absence of a condition. Usually used to signal that some other event or process is to take place.

FLOWCHART is a graphical depiction of the logical process that takes place within a computer program. Primarily used by programmers and systems analysts. Good programming theory dictates that flowcharting be used as a step to completely document a program. The flowchart should be developed before a program is actually coded so that any logic errors may be uncovered and corrected with a minimum of effort.

FORMAT is the form or condition that an item should be patterned after. See DATA FORMAT.

FORMATTING is a process whereby a blank disk is organized to allow the orderly storing of information for future retrieval and use.

HACKERS are sometimes viewed as social mutants. They are generally nothing more than fanatic programmers (see PROGRAMMER) that use computers to the exclusion of all else. To illustrate a point, hackers are to programmers what rough carpenters are to finish carpenters.

HARD COPY is printed output from a computer.

HARDWARE is the term that describes the physical circuits, chips, cards, and other items that make up a computer system.

HEADING is that portion of a report that identifies the contents of the report. It may include any information that would be of use to the user, such as titles, date, page numbers, column headings, etc.

HEXADECIMAL is the numbering system most often used in computers to represent equivalent binary data in a human-readable fashion. It is based on powers of 16, with each number position able to contain one of 16 digits, 0 through F.

INPUT is data that the computer receives from an outside source such as a keyboard, modem, or disk controller.

INTERFACE is the process of connecting or communicating between computer peripherals or between computer and humans. Interfacing may require special hardware (such as cables or modems) or software.

INTERPRETER is a program that translates each executable program step to machine code as it is time to execute it. Most versions of BASIC used on microcomputers are interpreter BASIC.

JUSTIFY means to line up all text to a certain column. It is usually used in connection with word-processing. Left-justify means all lines of text begin at a specific column. Right-justify means to force each line of text to end a specific column. Fill-justify means to insert spaces in individual lines of text to make sure that they begin and end at the same columns as other lines of text in the paragraph.

KEYBOUNCE is the rapid opening and closing of a keyboard relay after the original opening and closing. Many times this results in the appearance of two of the same characters when only one was pressed.

MAINFRAME is the largest of the three types of computers.

MENU is a series of choices. The choices represent the sum of all possible functions at the time that the menu is presented.

MICRO means very small. In the world of computers, Micro generally refers to the smallest member of the computer threesome. As a rule of thumb, if the computer can fit on a desk, it is usually a Micro.

MINI means small. As the middle members of the computer trilogy, minis have traditionally had less computing power than Mainframes, but slightly more than micros. Buying a mini may require mortgaging your kids and sub-letting your house, so many people have turned to micros to do more and more.

MODEM is a device usually used to connect your computer to the phone lines so that you can call other computers and run up your phone bill. Modem is a term that means MODulate - DEModulate. Many think that modems were created by the phone company to hook computerists and thus generate another source of revenue.

MODULATE means to convert electrical impulses to signal tones. These tones are often used to transfer information over telephone lines. See MODEM.

MOTHER BOARD is the term given to the main circuit board of a microcomputer. It comtains most, if not all, of the circuits and chips required to make the computer work.

NULL literally means nothing. When a string is equal to null, it has no length or no assignment.

PARSE means to interpret an instruction for syntactical format.

PEEK is a BASIC command that allows you to examine the value of a specific address within a computer's memory.

PERIPHERALS are pieces of equipment attached to a computer to help it do a specific task. Printers, disk drives, video monitors, and modems are all examples of peripherals.

POINTERS are markers used by a computer program to specify certain important limits or parameters.

POKE is a BASIC command that allows you to reassign the value of a specific computer memory address.

PRINTER is a device used to translate electronically stored information to a permanent, printed state.

PROGRAM is a series of instructions designed to make a computer perform a specific task. Programs tend to be intangible extensions of a programmer's subconscious mind.

PROGRAMMER is that breed of person that enters a program into a computer for hopefully error-free execution.

RAM is Random Access Memory—it is used by the computer to store information and programs. All data stored in RAM will disappear when the computer power is turned off.

RECORD is a collection of related fields. A group of records make up a file.

REPORT is an organized group of information used as functional output from a program. Hopefully a report will be of use to the person using the program.

ROM is Read-Only Memory. It is computer memory used to store information permanently. It does not go away when the power is turned off.

RUN TIME signifies that period of time during which a program is being executed. In a BASIC program, it is anytime after the RUN statement has been issued, and before control has been returned to the user.

SLOT refers to a connection on the inside of the Apple Computer which allows special computer devices to be attached to the computer. There are eight slots in the Apple II series of microcomputers. They are numbered 0 through 7 on the Apple II or II+, and 1 through 7 on the IIe. The eighth slot on the IIe is the 80-column/extra memory slot located in the middle of the Mother Board.

SOFTWARE refers to the programs (see PROGRAM) used on a computer. Basically, this is an interchangeable term for program.

STATIC is the great nemesis of computers. Static is that little bit of electricity that causes a spark when it touches a conductive surface after crossing a dry, carpeted room. It also has an untoward effect on computer data. The voltage spikes from static can erase data on disks or in computer chips.

STRING, short for "alphanumeric string," is a collection of characters. The notation "A$" is usually pronounced "A string".

SUBROUTINE is a group of computer instructions designed to collectively perform a set task. The subroutine may be "called" from any place necessary within a program, usually by a "GOSUB" command, and is terminated by the "RETURN" statement.

SYNTAX refers to the rules that govern the specific format that computer commands must follow.

TEXT FILE is a file (see FILE) saved to a disk as a series of ASCII characters, as opposed to saving as a series of compressed coded characters. Also called "data file" or "ASCII file".

TIME SHARING means to concurrently share CPU time between several different terminals. Usually accomplished on larger computers.

USER FRIENDLY means that the hardware or software "interfaces" well with a user. All this means is the degree of ease with which the system can be used. If a piece of software is described as being "user friendly", then it is

easy to learn and use. Usually user friendliness cannot be added after a system has been started. It needs to be designed in from the beginning.

UTILITY has two meanings. It can describe a class of programming tools which help a programmer become more productive by simplifying some task or helping him keep track of complex areas such as variables, line referencing, and program editing. Utility can also describe the functional value that a program has for a user. If a program has a "utilitarian value", then it is very useful.

Index

S

T

U

V

SAMS APPLE® BOOKS

Many thanks for your interest in this Sams Book about Apple II® microcomputing. Here are a few more Apple-oriented Sams products we think you'll like:

POLISHING YOUR APPLE®, Vol. 1

Clearly written, highly practical, concise assembly of all procedures needed for writing, disk-filing, and printing programs with an Apple II. Positively ends your searches through endless manuals to find the routine you need! By Herbert M. Honig. 80 pages, 5½ x 8½, comb. ISBN 0-672-22026-1. © 1982.
Ask for No. 22026 ..... **$4.95**

POLISHING YOUR APPLE®, Vol. 2

A second Apple II timesaver that guides intermediate-level programmers in setting up professional-looking menus, using effective error trapping, and making programs that run without the need for detailed explanations. Includes many sample routines. By Herbert M. Honig. 112 pages, 5½ x 8½, soft. ISBN 0-672-22160-8. © 1983.
Ask for No. 22160 ..... **$4.95**

APPLESOFT LANGUAGE (2nd Edition)

Quickly introduces you to Applesoft syntax and programming, including advanced techniques, graphics, color commands, sorts, searches, and more! New material covers disk operations, numbers, and number programming. Many usable routines and programs included. By Brian D. Blackwood and George H. Blackwood. 274 pages, 6 x 9, comb. ISBN 0-672-22073-3. © 1983.
Ask for No. 22073 ..... **$13.95**

APPLE® II APPLICATIONS

Presents a series of board-level interfacing applications you can modify if necessary to help you use an Apple II as a development system, a data-acquisition or control device, or for making measurements. Includes programs. By Marvin L. De Jong. 256 pages, 5½ x 8½, soft. ISBN 0-672-22035-0. © 1983.
Ask for No. 22035 ..... **$13.95**

DISKS, FILES, AND PRINTERS FOR THE APPLE® II

Provides you with basic-to-advanced details for using disks, files, and printers with an Apple II, including outstanding explanations for programming with sequential-access, random-access, and executive files. By Brian D. Blackwood and George H. Blackwood. 216 pages, 6 x 9, comb. ISBN 0-672-22163-2. © 1983.
Ask for No. 22163 ..... **$15.95**

THE APPLE® II CIRCUIT DESCRIPTION

Provides you with a detailed circuit description for all revisions of the Apple II and Apple II+ motherboard, including the keyboard and power supply. Highly valuable data that includes timing diagrams for major signals, and more. By Winston D. Gayler. 176 pages plus foldouts, 8½ x 11, comb. ISBN 0-672-21959-X. © 1983.
Ask for No. 21959 ..... **$22.95**

INTERMEDIATE LEVEL APPLE® II HANDBOOK

Provides you with a nicely paced transition from Integer BASIC into machine- and assembly-language programming with the Apple II. Covers text display, video POKEs, graphics, using machine language with BASIC, memory addresses, debugging, and more. By David L. Heiserman. 328 pages, 6 x 9, comb. ISBN 0-672-21889-5. © 1983.
Ask for No. 21889 ..... **$16.95**

APPLE® FORTRAN

Only fully detailed Apple FORTRAN manual on the market! Ideal for Apple programmers of all skill levels who want to try FORTRAN in a business or scientific program. Many ready-to-run programs provided. By Brian D. Blackwood and George H. Blackwood. 240 pages, 6 x 9, comb. ISBN 0-672-21911-5. © 1982.
Ask for No. 21911 ..... **$14.95**

APPLE® II ASSEMBLY LANGUAGE

Shows you how to use the 3-character, 56-word vocabulary of Apple's 6502 to create powerful, fast-acting programs! For beginners or those with little or no assembly language programming experience. By Marvin L. De Jong. 336 pages, 5½ x 8½, soft. ISBN 0-672-21894-1. © 1982.
Ask for No. 21894 ..... **$15.95**

ENHANCING YOUR APPLE® II — Vol. 1

Shows you how to mix text, LORES, and HIRES anywhere on the screen, how to open up whole new worlds of 3-D graphics and special effects with a one-wire modification, and more. Tested goodies from a trusted Sams author! By Don Lancaster. 232 pages, 8½ x 11, soft. ISBN 0-672-21846-1. © 1982.
Ask for No. 21846 ..... **$17.95**

CIRCUIT DESIGN PROGRAMS FOR THE APPLE® II

Programs quickly display "what happens if" and "what's needed when" as they apply to periodic waveform, rms and average values, design of matching pads, attenuators, and heat sinks, solution of simultaneous equations, and more. By Howard M. Berlin. 136 pages, 8½ x 11, comb. ISBN 0-672-21863-1. © 1982.
Ask for No. 21863 ..... **$15.95**

APPLE® INTERFACING

Brings you real, tested interfacing circuits that work, plus the necessary BASIC software to connect your Apple to the outside world. Lets you control other devices and communicate with other computers, modems, serial printers, and more! By Jonathan A. Titus, David G. Larsen, and Christopher A. Titus. 208 pages, 5½ x 8½, soft. ISBN 0-672-21862-3. © 1981.

Ask for No. 21862 ..... **$11.95**

INTIMATE INSTRUCTIONS IN INTEGER BASIC

Explains flowcharting, loops, functions, graphics, variables, and more as they relate to Integer BASIC. Used with *Applesoft Language* (No. 22073), it gives you everything you need to program BASIC with your Apple II or Apple II Plus. By Brian D. Blackwood and George H. Blackwood. 160 pages, 5½ x 8½, soft. ISBN 0-672-21812-7. © 1981.

Ask for No. 21812 ..... **$8.95**

MOSTLY BASIC: APPLICATIONS FOR YOUR APPLE® II, BOOK 1

Twenty-eight debugged, fun-and-serious BASIC programs you can use immediately on your Apple II. Includes a telephone dialer, digital stopwatch, utilities, games, and more. By Howard Berenbon. 160 pages, 8½ x 11, comb. ISBN 0-672-21789-9. © 1980.

Ask for No. 21789 ..... **$13.95**

MOSTLY BASIC: APPLICATIONS FOR YOUR APPLE® II, BOOK 2

A second gold mine of fascinating BASIC programs for your Apple II, featuring 3 dungeons, 11 household programs, 6 on money or investment, 2 to test your ESP level, and more — 32 in all! By Howard Berenbon. 224 pages, 8½ x 11, comb. ISBN 0-672-21864-X. © 1981.

Ask for No. 21864 ..... **$12.95**

SAMS SOFTWARE FOR THE APPLE®

FINANCIAL PLANNING WITH VISICALC® AND THE APPLE® II

Automatically sets up your VisiCalc spreadsheet to perform 16 different calculations commonly needed in business and financial planning, and lets you compare as many as four possibilities. Works with 80-column board if you have one. You'll need VisiCalc, 64K RAM, and one disk drive. ISBN 0-672-29059-6.

Ask for No. 29059 ..... **$79.95**

FINANCIAL PLANNING WITH MULTIPLAN™ AND THE APPLE® II

Same as *Financial Planning with VisiCalc*, except works with Multiplan spreadsheet, 64K RAM, and one disk drive. ISBN 0-672-29058-8.

Ask for No. 29058 ..... **$79.95**

MONEY TOOL

Helps you manage income, expenses, and tax information for home or small business. Can reconcile checking, provide simple reports, and more. By Herb Honig. Takes 48K RAM, Applesoft in ROM, and one disk drive. ISBN 0-672-26113-8.

Ask for No. 26113 ..... **$59.95**

FINANCIAL FACTS

Instantly computes the majority of data you'll commonly need in personal and small-business financial management, and prints out the major factors. By Ed Hanson. Takes 48K RAM, Applesoft in ROM, and one disk drive. ISBN 0-672-26099-9.

Ask for No. 26099 ..... **$59.95**

INSTANT RECALL

Friendly, unconventional, and instantaneous data handler. Each free-form, alphanumeric screenful you enter is an 840-character page you can edit, file, or print out as it appears. By Charles R. Landers. Takes 48K RAM, Applesoft in ROM, and one disk drive. ISBN 0-672-26097-2.

Ask for No. 26097 ..... **$59.95**

PEN-PAL

Sophisticated, powerful, affordable word processor. Provides block movement, line deletion, character and text insertion, and more. Takes 48K RAM, Applesoft in ROM, and one disk drive. ISBN 0-672-26115-4.

Ask for No. 26115 ..... **$59.95**

HELLO CENTRAL !

Versatile, menu-controlled terminal program you can use with any compatible modem to communicate with networks and other computers. Has built-in text editor, auto dialing, much more. By Bruce Kallick. Takes 48K RAM, Applesoft in ROM, one disk drive, and modem. ISBN 0-672-26081-6.

Ask for No. 26081 ..... **$99.95**

You can usually find these Sams products at better computer stores, bookstores, and electronic distributors nationwide.

If you can't find what you need, call Sams at 800-428-3696 toll-free or 317-298-5566, and charge it to your MasterCard or Visa account. Prices subject to change without notice.

For a free catalog of all Sams Books available, write P.O. Box 7092, Indianapolis, IN 46206.

TO THE READER

Sams Computer books cover Fundamentals — Programming — Interfacing — Technology written to meet the needs of computer engineers, professionals, scientists, technicians, students, educators, business owners, personal computerists and home hobbyists.

Our Tradition is to meet your needs and in so doing we invite you to tell us what your needs and interests are by completing the following:

1. I need books on the following topics:

_____

_____

_____

_____

_____

_____

2. I have the following Sams titles:

_____

_____

_____

_____

_____

_____

3. My occupation is:

_____ Scientist, Engineer
_____ Personal computerist
_____ Technician, Serviceman
_____ Educator
_____ Student
_____ D P Professional
_____ Business owner
_____ Computer store owner
_____ Home hobbyist
Other _____

_____

Name (print) _____

Address _____

City _____ State _____ Zip _____

Mail to: **Howard W. Sams & Co., Inc.**
Marketing Dept. #CBS1/80
4300 W. 62nd St., P.O. Box 7092
Indianapolis, Indiana 46206

22208

DISKETTE

Please send me _____ copies of the **DOS 3.3 COMPANION DISKETTE** to *BASIC Tricks for the Apple,* at $9.95 each. I understand this disk is fully copyable for my personal use only.

Make check or money order payable to Discovery Software.

Allow 3-4 weeks for processing. C.O.D. orders will be charged an additional C.O.D. fee.

☐ I enclose check for $ _____

☐ Please send C.O.D.

NAME _____

ADDRESS _____

CITY _____

STATE _____ ZIP _____

TELEPHONE (____) _____

Please, no purchase orders. We also cannot ship to a foreign address.

FROM

PLACE
POSTAGE
HERE

DISCOVERY SOFTWARE
P.O. BOX 68821
INDIANAPOLIS, IN 46268

SAMS

BASIC Tricks For The Apple®

Go one step beyond the fundamentals of BASIC programming on the Apple and learn to perform the "tricks" which will make your programs more useful and efficient. In clear, concise fashion, this book guides you through the logic, creation and integration of some 35 routines—including routines designed to:

- Give reports a professional appearance with rounded and aligned numbers, column headings, and centered or justified lines.
- Allow you to input and print times and dates in a standard format and use them in program calculations.
- Create menu screens for your programs.
- Sort array variables by any one of four different methods and compare the time required by each.
- Convert names or other input containing upper and lower case characters to all upper or all lower case.
- Write data to and read data from both random access and sequential files.
- Simplify programming error detection and correction.

Howard W. Sams & Co., Inc.
4300 West 62nd Street, Indianapolis, Indiana 46268 U.S.A.

$8.95/22208

ISBN: 0-672-22208-6